Handlers for Non-Monadic Computations (Extended Version)

Ruben P. Pieters
Tom Schrijvers
Exequiel Rivas

Report CW 713, March 2018

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Handlers for Non-Monadic Computations (Extended Version)

Ruben P. Pieters
Tom Schrijvers
Exequiel Rivas

Report CW 713, March 2018

Department of Computer Science, KU Leuven

Abstract

Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers are somewhat limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

Handlers for Non-Monadic Computations (Extended Version)

Ruben P. Pieters
KU Leuven
ruben.pieters@cs.kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@cs.kuleuven.be

Exequiel Rivas
CONICET – UNR
rivas@cifasis-conicet.gov.ar

ABSTRACT

Algebraic effects and handlers are a convenient method for structuring monadic effects with primitive effectful operations and separating the syntax from the interpretation of these operations. However, the scope of conventional handlers are somewhat limited as not all side effects are monadic in nature.

This paper generalizes the notion of algebraic effects and handlers from monads to generalized monoids, which notably covers applicative functors and arrows. For this purpose we switch the category theoretical basis from free algebras to free monoids. In addition, we show how lax monoidal functors enable the reuse of handlers and programs across different computation classes, for example handling applicative computations with monadic handlers.

CCS CONCEPTS

• **Theory of computation** → **Functional constructs**; • **Software and its engineering** → **Functional languages**;

KEYWORDS

algebraic effects, effect handlers, generalized monoids, monads, applicative functors, arrows

1 INTRODUCTION

Since their introduction to purely functional programming, monads [12, 16] have monopolized modeling computational effects. This changed with the proposal of new classes of computation: applicative functors [11] and arrows [4], which capture types of side effects amenable to static analysis at the cost of expressiveness.

In a separate development, algebraic effects and handlers [13] were created as a more convenient formulation of monadic effects and programs. Their success is largely due to their easier integration with impure functional and imperative languages to enable user-defined effects. This approach encodes effects as operations represented by the signature of an algebraic theory. The semantics of these effects is represented by an interpretation for the operations.

Although the conventional handlers capture monadic effects well, other computation classes such as applicative functors and arrows are not covered. To remedy this situation, Lindley [7] presented a language design supporting handlers for the classic triad of effects: monad, arrow and applicative. This is backed by a type system verifying the class of expressed computations. However, Lindley's exposition lacks an extension of the category theoretical underpinnings, introduced by Plotkin and Pretnar.

This work aims to provide this extension by reviewing the definition of handlers to include non-monadic computations, notably applicative functors and arrows. For this purpose we leverage the framework of Rivas and Jaskelioff [14] which characterizes the triad of effects in terms of generalized monoids. This is used to replace the conventional free algebra approach, with handling rules based on the unique algebra homomorphism, by a free monoid approach, with handling rules based on the unique monoid homomorphism.

Specifically our contributions are:

- We present a generic framework to derive handlers for monoids in monoidal categories.
- We give a derivation of handlers for the classes of applicative, arrow and monadic effects. Since the derived monadic handlers are equally expressive as the conventional free algebra handlers, we see the monoidal handlers as an extension to the free algebra handlers.
- We present a method for reusing handlers and programs, by employing an adjunction with a lax monoidal functor between the relevant monoidal categories.

Section 2 introduces and motivates the concept of non-monadic handlers. Section 3 introduces the relevant category theoretic background related to algebraic effects and handlers, presenting them as free algebras. Section 4 derives these handlers from the perspective of free monoids. Section 5 derives applicative and arrow handlers from the idea of free monoids. Section 6 shows an approach to reuse handlers and computations across different monoidal categories. Section 7 presents and discusses related work.

This paper comes with a comprehensive appendix of proofs, establishing the validity of our contributions.

2 MOTIVATION

Plotkin and Pretnar [13] introduce algebraic effects and effect handlers, which we refer to as monadic handlers. This is the conventional approach for languages implementing handlers.

This section elaborates on the difference between monadic and non-monadic handlers and presents a use case involving non-monadic handlers.

2.1 Classes of Computations

Lindley [7] illustrates three classes of computations: monadic, arrow and applicative computations. These three classes are illustrated below.

Monadic Computations. Monadic computations have both dynamic *control* and *data* flow. The former implies a dependency of subsequent operations on results of previous operations. The latter implies that input to operations is dependent on results of previous operations. In the following example, result `x` influences both which operation takes place next, and the parameter to those operations. Thus, it is a monadic computation.

```
do x <- op1
   if x                                (control flow)
     then op2 (f x)                       (data flow)
     else op3 (g x)                       (data flow)
   return x
```

Where each $\texttt{op}_i$ is effectful and `f` and `g` are pure.

Arrow Computations. Arrow computations only contain dynamic data flow, meaning only operation inputs can depend on previous results. In the example, `x` influences the input to $\texttt{op}_2$, but `x` does not influence which operation is executed.

```
do x <- op1
   op2 (f x)                           (data flow)
   op3 (g x)                           (data flow)
   return x
```

Applicative Computations. Applicative computations contain neither dynamic control nor dynamic data flow. They can be seen as a static list of operations to be executed, while computing a final value from their results. A simple example:

```
do x <- op1
   op2 c
   op3 d
   return x
```

2.2 Monadic Handlers

The handler approach to computational effects separates introduction of effects from their interpretation. Calling an operation introduces an effect, while a handler interprets these operations.

The upcoming examples utilize two operations. Operation `read` takes a location ($\mathbb{L}$) parameter and returns a number ($\mathbb{N}$). Operation `write` takes a location and a number and returns the unit value `()`. This is summarized in these signatures:

```
read : 𝕃 -> ℕ     write : (𝕃,ℕ) -> ()
```

Calling these operations creates a computation out of them:

```
comp = do
  write ("loc1",1)
  write ("loc2",1)
  write ("loc1",2)
  x <- read "loc1"
  y <- read "loc2"
  return (x + y)
```

Where location parameters are surrounded by `""` quotes.

This is merely a description of an abstract computation until interpreted by a handler. The handler gives meaning to each operation in its operation clauses, and a final translation from a fully evaluated computation in its value clause.

Interpretation to `IO`. In code, this looks as follows:

```
handlerIO = handler
| val (x: A) -> return x            (value clause)
| write (p: (𝕃,ℕ),k: () -> IO A)
    -> do storeAt p
          k ()             (write operation clause)
| read (p: 𝕃,k: ℕ -> IO A)
    -> do n <- retrieveFrom p
          k n               (read operation clause)
```

This handler interprets to the `IO A` type signifying a computation possibly returning an `A` which might execute arbitrary side effects. We omit the definitions of `storeAt` and `retrieveFrom`, they could for example write to or read from a database. We want to focus on the parameters which are available in the value and operation clauses.

The value clause triggers on an evaluated computation without any operations, it takes a value of type `A` as parameter `x` and evaluates to a value of type `IO A`.

The operation clause triggers when the evaluated computation is an operation with a continuation. It takes the input arguments as parameter `p` and the continuation as parameter `k`. The former contains all data passed to the operation. The latter captures a resumption point, which resumes the computation where the operation was called and introduces a result. The computation does not resume if the continuation parameter is not invoked, resulting in behavior similar to exceptions. The operation clause evaluates to an `IO A` value.

Interpreting with `handlerIO` gives a value of type `IO ℕ`:

```
handle comp with handlerIO ==
do storeAt ("loc1",1)
   storeAt ("loc2",1)
   storeAt ("loc1",2)
   x <- retrieveFrom "loc1"
   y <- retrieveFrom "loc2"
   return (x + y)
```

Interpretation to `Map`. Computation `comp` can be interpreted differently by using another handler. For example, creating a description of the final state of all locations.

This first attempt interprets to the `Map -> Map` type, where `Map` stores number values indexed by locations. The `Map` type supports operations such as `storeInMap : Map -> (𝕃,ℕ) -> Map` and `getFromMap : Map -> 𝕃 -> ℕ` to store and retrieve values respectively. The following handler implements the creation of a summarizing map:

```
handlerMap = handler
| val (x: A) -> λ(m: Map). m
```

```
| write (p: (𝕃,ℕ),k: () -> (Map -> Map))
    -> λ(m: Map). let m' = storeInMap m p
                  in  k () m'
| read (p: 𝕃,k: ℕ -> (Map -> Map))
    -> λ(m: Map). let n = getFromMap m p
                  in k n m
```

We interpret by evaluating `handle comp with handlerMap`, resulting in a value with type `Map -> Map`. By passing the empty map, `emptyMap`, the resulting description map is obtained. Thus, `(handle comp with handlerMap) emptyMap` evaluates to `Map ("loc1" ↦ 2, "loc2" ↦ 1)`.

Limitation of Monadic Handlers. However, this handler does not exactly do what we intended, which is to create a static map by analyzing the computation and returning a summary of all final updates. The result of our handler requires an initial `Map`, on which the summarizing map depends!

The necessity of this dependency might become clear by considering the following computation:

```
readWrite = do
  x <- read "loc1"
  write ("loc2",x)
  return x
```

How can a summarizing map be determined without ever passing an initial map? This is not possible since the final value in `"loc2"` depends on the value in `"loc1"`.

The interface of monadic handlers is limited, they must be able to handle *all* monadic computations. Since there is no way to calculate a static summarizing map in general, this is not possible to express with a monadic handler.

This results in the following failing attempt to create an interpretation to a `Map` type with a monadic handler. The `read` clause does not have a sensible implementation for our intent, it requires us to know which number ℕ will be returned from the `read` operation to obtain the currently built map.

```
handlerSumMap = handler
| val (x: A) -> emptyMap
| write (p: (𝕃,ℕ),k: () -> Map)
    -> setDefault (k ()) p
| read (p: 𝕃,k: ℕ -> Map)
    -> ?
```

Where function `setDefault` sets the value in a location if it is not yet present.

2.3 Non-Monadic Handlers

Non-Monadic handlers solve this restricted interface because they are applicable only to strict subsets of monadic computations [9]. Applicatives, to which computation `comp` belongs, is one of those subsets. Interpretation to a more restrictive class liberates the handler, allowing a much broader space of interpretations. This results in an interpretation to a `Map` type, which was not possible with monadic handlers. This handler can be sketched as:

```
handlerSumMap = handler
| val (x: A) -> emptyMap
| write (p: (𝕃,ℕ),k: _,f: Map)
    -> setDefault f p
| read (p: 𝕃,k: _,f: Map)
    -> f
```

For the sake of clarity it simplifies the proposed notation for applicative handlers used in Section 5. The type of the uninvoked continuation `k` is left implicit here by using an underscore, the actual type of `k` is explained in Section 5.

Evaluating `handle comp with handlerSumMap` results in:

```
setDefault (
  setDefault (
    setDefault (
      emptyMap
    ) ("loc1",2)
  ) ("loc2",1)
) ("loc1",1)
```

Which is the map `Map ("loc1" ↦ 2, "loc2" ↦ 1)`.

Attempting to handle a computation with an inappropriate handler, for example `handle readWrite with handlerSumMap` should result in a runtime or, preferably, a type error.

This summarizing map example illustrates a simple use case of analysis with a non-monadic handler. More complicated analysis includes parallelizing/batching operations, calculating a heat map of operations/parameters such as location, or other calculations on statically available information.

3 BACKGROUND

This section introduces the necessary background on which the remainder of the paper is based. We assume basic familiarity with common category theoretical concepts such as functors, natural transformations and monads.

3.1 Notational Conventions

We highlight some of the more specific notation here.

Do-Notation. The intent of the sugared notation for computations is to be consistent with Haskell's `do`-notation.

Category. We reserve $\mathscr{C}$ to mean the category of the programming language under consideration, with types as objects and functions between those types as morphisms, even if strictly speaking this would not form a category [3].

Morphisms. Components of natural transformations will usually have a subscript mentioning their naturality, e.g. $id_A : A \to A$ is natural in A. Identity morphisms are denoted as the more compact $A : A \to A$ instead of $id_A : A \to A$.

(Co-)Products. We use $A \times B$ to denote products, in $\mathscr{C}$ this represents the tuple type `(A,B)`. We use $A + B$ to denote coproducts, and $[f, g]$ to denote the unique morphism $A+B \to X$ constructed from $f : A \to X$ and $g : B \to X$.

Exponential Objects. We use A^B to denote the exponentiation of A with B. In $\mathscr{C}$ exponential A^B is the function type `B -> A`.

Algebra of a Functor. An F-algebra with *carrier* A and *action* b is denoted by $\langle A, b : FA \to A\rangle$.

(Co-)Ends. We denote ends as $\int_A F(A, A)$ and co-ends as $\int^A F(A, A)$, for a bifunctor $F : \mathscr{A}^{op} \times \mathscr{A} \to \mathscr{B}$. In $\mathscr{C}$, ends correspond to universal type quantification `∀A. F(A,A)`, while co-ends correspond to existential type quantification `∃A. F(A,A)`. Usually the type quantifier ∀ is omitted when it is clear from context.

3.2 Algebraic Effects and Handlers

Plotkin and Pretnar's definition of algebraic effects and handlers consists of two parts: the *operations*, which introduce effects, and the *handlers*, which interpret them [13].

Operations as Functors. Operations, like `get` and `put`, are abstracted by endofunctors of the form $\Sigma_i = P_i \times -^{N_i}$, where N_i is the *arity* of the operation, and P_i are the *parameters* of the operation. The former refers to the type of values which the operation introduces into the computation, the latter refers to the type of values which the operation takes as input. For example, given signatures for `get` and `put`:

```
get : () -> S      put : S -> ()
```

Where `S` is the state type. `get` and `put` are represented by $\Sigma_{\texttt{get}} = \texttt{()} \times -^{S}$ and $\Sigma_{\texttt{put}} = S \times -^{\texttt{()}}$ respectively.

The representation of all operations is obtained by constructing the coproduct of their respective functors $\Sigma = (P_0 \times -^{N_0}) + \ldots + (P_n \times -^{N_n})$. For example, Σ for $\{\texttt{get}, \texttt{put}\}$ is equal to $\Sigma = \texttt{()} \times -^{S} + S \times -^{\texttt{()}}$.

Operation Clauses as Σ-Algebras. Each of the operation clauses in a monadic handler gives an algebra for Σ_i, where i is the operation of interest. For example, the clause

```
| put (p: S,k: () -> B) -> b: B                    (c_put)
```

represents $\langle \texttt{B}, c_{\texttt{put}} : \Sigma_i \texttt{B} \to \texttt{B}\rangle$, a Σ_i-algebra. The function defined by the clause, is named in brackets on the right. So, $c_{\texttt{put}}$ is the function `λ(p: S,k: () -> B). b`.

The combination of all operation clauses

```
| op_i (p_i: P_i,k_i: N_i -> B) -> b: B            (c_i)
```

form the Σ-algebra $\langle \texttt{B}, c = [c_0, \ldots, c_n] : \Sigma \texttt{B} \to \texttt{B}\rangle$.

The value clause

```
| val (a: A) -> b: B                              (v)
```

defines the function $v : \texttt{A} \to \texttt{B} = \lambda\texttt{(a: A). b}$.

Handling Rules as Equations. Given a handler `h`:

```
h = handler
| val (a: A) -> ...: B                            (v)
| op_i (p_i: P_i,k_i: N_i -> B) -> ...: B         (c_i)
```

Evaluating `handle x with h` requires both the value and operation rules. The *value rule* triggers when no operations are left in a fully evaluated `x`, usually in the form of `return y`. The result is defined as:

```
handle (x: A) with h = v x
```

The *operation rule* triggers when the evaluated computation is an operation $\texttt{op}_i$. The result is defined as:

```
handle (op_i (p: P_i,◇: N_i -> Σ*A)) with h
= c_i p (λn. handle (◇ n) with h)
```

Where the structure $\Sigma^* A$ represents a computation built from operations present in Σ, which aims to return a value of type A. The parameter $\diamond$ denotes the *continuation* of the operation call. For example, in

```
example = do
  x <- get ()
  put x
  return x
```

The continuation $\diamond$ of `get ()` is the function

```
λ(x: S). do
  put x
  return x
```

Syntax Constructors. Desugaring of the `do`-notation is possible with the constructors $\texttt{val}_A$ and $\texttt{op}_A$. The former $\texttt{val}_A$ embeds an evaluated value of type A into $\Sigma^* A$, and the latter $\texttt{op}_A$ embeds an operation into $\Sigma^* A$. With these constructors, the computation `example` is represented as:

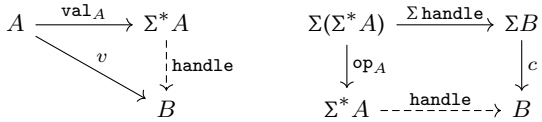
```
example =
ops (get ((),(λ(x: S).
  ops (put (x,(λ(_: ()).
    vals x
  )))
)))
```

These two constructors enable expressing the handling rules as pointfree equations. The pointfree value and operation rules are respectively: $\texttt{handle} \circ \texttt{val}_A = v$ and $\texttt{handle} \circ \texttt{op}_A = c_i \circ \Sigma\,\texttt{handle}$.

Handlers for Free Algebras. The elements from the previous section enable viewing monadic handlers as free algebras:

Definition 3.1 (Free Σ-Algebra). A free Σ-algebra on A in $\mathscr{C}$ consists of an object $\langle \Sigma^* A, \texttt{op}_A : \Sigma(\Sigma^* A) \to \Sigma^* A\rangle$ in Σ-$Alg(\mathscr{C})$ together with a morphism $\texttt{val}_A : A \to \Sigma^* A$ in $\mathscr{C}$ such that for any $\langle B, c : \Sigma B \to B\rangle$ in Σ-$Alg(\mathscr{C})$ and morphism $v : A \to B$ in $\mathscr{C}$, there exists a unique morphism $\texttt{handle} : \langle \Sigma^* A, \texttt{op}_A\rangle \to \langle B, c\rangle$ in Σ-$Alg(\mathscr{C})$ with $\texttt{handle} \circ \texttt{val}_A = v$.

The diagrams for the conditions are:



The diagram on the left-hand side is the condition mentioned in the definition and corresponds to the value rule equation. The diagram on the right-hand side is the condition for a morphism in Σ-$Alg(\mathscr{C})$, namely a Σ-algebra homomorphism, and corresponds to the operation rule equation.

3.3 Monoids in Monoidal Categories

Rivas and Jaskelioff [14] present a framework for different classes of side effects as (generalized) monoids in various monoidal categories. We reintroduce the relevant definitions relating to monoidal categories in the following paragraphs.

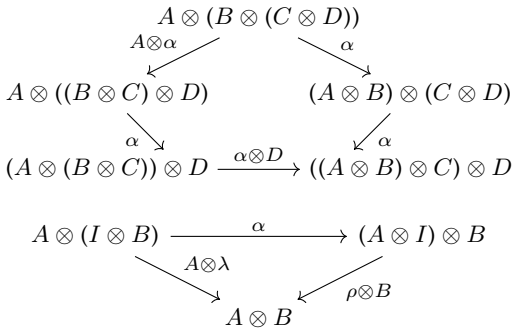
Monoidal Category. A monoidal category is a category which contains a notion of monoids generalizing the monoids in *Set*.

Definition 3.2 (Monoidal Category). A monoidal category is a tuple $(\mathscr{D}, \otimes, I, \alpha, \lambda, \rho)$, consisting of

a) a category $\mathscr{D}$
b) a bifunctor $\otimes : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ (also called the *tensor*)
c) a designated object I of $\mathscr{D}$
d) three natural isomorphisms

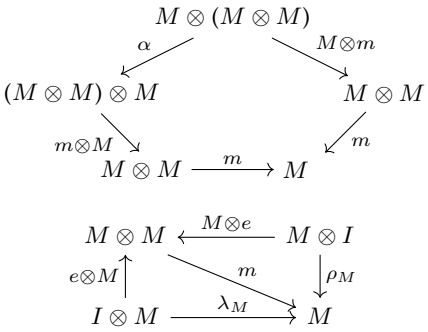
$$\begin{aligned} \alpha_{A,B,C} &: A \otimes (B \otimes C) \to (A \otimes B) \otimes C \\ \lambda_A &: I \otimes A \to A \\ \rho_A &: A \otimes I \to A \end{aligned}$$

such that $\lambda_I = \rho_I$ and the following diagrams commute:



Monoid in Monoidal Category. A monoid in a monoidal category is a generalization of monoids in *Set*. From this point on, whenever we mention *monoid*, we mean the more general concept *monoid in monoidal category*.

Definition 3.3 (Monoid in Monoidal Category). A monoid in a monoidal category is a tuple (M, e, m) where M is an object in a monoidal category $(\mathscr{D}, \otimes, I, \alpha, \lambda, \rho)$. The multiplication $m : M \otimes M \to M$ and the unit $e : I \to M$ are morphisms in $\mathscr{D}$ such that the following diagrams commute:



Exponentials for Monoidal Categories. The characterizing isomorphism of exponentials can be generalized with tensors instead of products. This results in the isomorphism: $\lfloor - \rfloor : \mathscr{D}(X \otimes B, A) \cong \mathscr{D}(X, A^B) : \lceil - \rceil$. Thus, the evaluation morphism can be generalized to $ev_A : A^B \otimes B \to A = \lceil A^B \rceil$.

Example 3.4. The main examples of monoidal categories we consider are:

(1) The category of endofunctors, *End*, with functor composition $(F \circ G)A = F(GA)$ as tensor and the identity functor *Id* as designated object. This monoidal category is called *strict* since α, λ and ρ are identities. Monoids in this monoidal category are known as monads.
(2) The category of endofunctors *End*, with Day convolution $(F \star G)A = \int^Z F(Z \to A) \times GZ$ as tensor and the identity functor *Id* as designated object. Monoids in this monoidal category are known as applicative functors.
(3) The category of strong profunctors, *SPro*, with profunctor composition $(P \otimes Q)(A, B) = \int^Z P(A, Z) \times Q(Z, B)$ as tensor and the *Hom* profunctor as designated object. Monoids in this monoidal category are known as arrows.

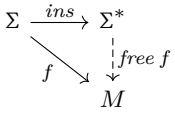
Category of Monoids. For a monoidal category $(\mathscr{D}, \otimes, I, \alpha, \lambda, \rho)$, we have the category of monoids $Mon(\mathscr{D})$ which consists of all monoids (M, e, m) in that monoidal category and monoid homomorphisms between them.

4 HANDLERS FOR FREE MONOIDS

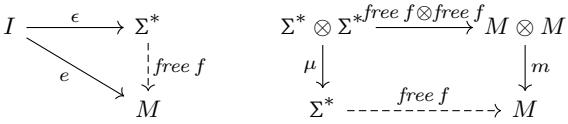
In order to include other classes of effects, this section derives a notion of handlers for free monoids. We begin by recalling the definition of the free monoid on an object:

Definition 4.1 (Free Monoid). A free monoid on an object Σ in a category $\mathscr{D}$ consists of an object $(\Sigma^*, \epsilon, \mu)$ in $Mon(\mathscr{D})$ together with a morphism $ins : \Sigma \to \Sigma^*$ in $\mathscr{D}$ such that for any (M, e, m) in $Mon(\mathscr{D})$ and morphism $f : \Sigma \to M$ in $\mathscr{D}$, there exists a unique morphism $free\, f : (\Sigma^*, \epsilon, \mu) \to (M, e, m)$ in $Mon(\mathscr{D})$ with $free\, f \circ ins = f$.

The condition represented in a diagram is:



The diagrams corresponding to the monoid homomorphism condition, a morphism in $Mon(\mathscr{D})$, are:



More categorically, free monoids arise as the left adjoint to the forgetful functor from $Mon(\mathscr{D})$ to $\mathscr{D}$.

4.1 Monoidal Handlers

Monoidal Basis. We can interpret the carrier Σ^* of a free monoid as the syntax of computations. Instead of using `val` and `op` from Section 3.2, we can construct programs in this *monoidal* syntax from the constructors ϵ, μ and *ins* provided by the free monoid.

Example 4.2. In the following example, we set the category $\mathscr{D}$ as $End(\mathscr{C})$, and our monoidal category as monads on $\mathscr{C}$. From the general monoidal syntax (left) follows the specialized syntax (right) for this setting:

$$\begin{array}{llll} \epsilon & : I \to \Sigma^* & \epsilon_A & : A \to \Sigma^* A \\ \mu & : \Sigma^* \otimes \Sigma^* \to \Sigma^* & \mu_A & : \Sigma^*(\Sigma^* A) \to \Sigma^* A \\ ins & : \Sigma \to \Sigma^* & ins_A & : \Sigma A \to \Sigma^* A \end{array}$$

The constructor ϵ_A is similar to $\texttt{val}_A$: it embeds a value into a computation. Constructor ins_A embeds an operation into a computation. To embed an operation returning another computation, of type $\Sigma(\Sigma^* A)$, we use naturality of ins_A to obtain $ins_{\Sigma^* A} : \Sigma(\Sigma^* A) \to \Sigma^*(\Sigma^* A)$. Lastly, μ_A converts a computation returning a computation into a flat computation.

Consider the `example` computation again:

```
example = do
  x <- get ()
  put x
  return x
```

Which was constructed using `val`/`op` as:

```
example =
opS (get ((),(λ(x: S).
  opS (put (x,(λ(_: ()).
    valS x
  )))
)))
```

The same program can be constructed with the monoidal constructors, namely $\epsilon/\mu/ins$:

```
μS (insΣ*S (get ((),(λ(x: S).
  μS (insΣ*S (put (x,(λ(_: ()).
    ϵS x
  ))))
))))
```

Monoidal Handler. The unique monoid morphism *free* f, induced by a morphism $f : \Sigma \to M$, is the handler construct for monoidal programs. These monoidal handlers are defined by a clause for the constructors ϵ and μ, and other clauses for each operation; each clause evaluates programs to a monoid (M, e, m). The unit $e : I \to M$ and multiplication $m : M \otimes M \to M$ of this monoid is defined by the clauses for the ϵ and μ constructor respectively. This definition is expected to satisfy the monoid laws, but the notation does not enforce this. All operation clauses are combined to define the morphism $f : \Sigma \to M$, which interprets the constructor *ins*.

In the example, monads on $\mathscr{C}$, the clauses to define are:

```
mh = mhandler
| ϵ (a: A) -> ...: MA                    (eA)
| μ (mma: M(MA)) -> ...: MA               (mA)
| opi (pi: Pi,k: Ni -> A) -> ...: MA      (fA)
```

Notably, the handling construct is named `mhandler`, as opposed to `handler`, to signify a monoidal handler.

The evaluation rules are determined by the conditions in the free monoid definition.

The ϵ rule is similar to the `val` rule.

```
mhandle (x: A) with mh = eA x
```

The μ rule forwards the handling to both Σ^* structures, and then combines the result using multiplication m from the monoid (M, e, m).

```
mhandle (s: Σ*(Σ*A)) with mh
= mA (mhandle (Σ*(mhandle _ with mh) s) with mh)
```

The *ins* rule interprets an operation $\texttt{op}_i$ with a function f.

```
mhandle (opi (p: Pi) (◇: Ni -> A)) with mh
= fA (p,◇)
```

Example 4.3. As an example, a handler implementation to interpret `get`/`put` into state passing functions `S -> S×A` is given below.

```
mStateMon = mhandler
| ϵ (a: A) -> λ(s:S). (s,a)
| μ (mma: S -> ((S,S) -> (S,A)))
    -> λ(s:S). let (s':S,ma:S -> (S,A)) = mma s
               in ma s'
| get (_: (),k: S -> A) -> λ(s:S). ((),(k s))
| put (p: S,k: () -> A) -> λ(s:S). (p,k ())
```

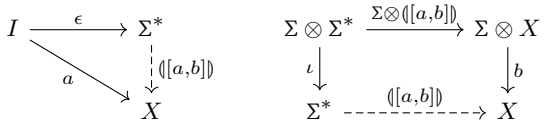
4.2 Inductive Handlers

Initial Algebra Basis. In the presence of exponentials, the free monoid can be represented constructively as the initial algebra of the $I+\Sigma\otimes-$ functor [14]. Concretely, this gives us an alternative set of constructors: $\epsilon : I \to \Sigma$ and $\iota : \Sigma \otimes \Sigma^* \to \Sigma^*$. These morphisms are the two elements of the initial algebra $[\epsilon, \iota] : I + \Sigma \otimes \Sigma^* \to \Sigma^*$.

Using this alternative syntax, the `example` computation is constructed as:

```
example =
ιS (get ((),(λ(x: S).
  ιS (put (x,(λ(_: ()).
    ϵS x
  )))
)))
```

Initial Algebra Handler. This alternative basis derives its handler by using the unique algebra homomorphism from the initial algebra. This unique morphism is denoted $(\!|[a,b]|\!) : \Sigma^* \to X$ for a morphism $a : I \to X$ and $b : \Sigma \otimes X \to X$. It is the unique morphism for which the following diagrams commute:



This results in a new handling construct `ihandler`. For the monads on $\mathscr{C}$ example, it requires the following clauses:

```
ih = ihandler
| ϵ (a: A) -> ...: XA                    (e_A)
| op_i (p: P_i,k: N_i -> XA) -> ...: XA   (g_A)
```

Which has no laws attached.

The evaluation rules follow from the algebra homomorphism conditions.

The ϵ rule is unchanged:

```
ihandle (x: A) with ih = e_A x
```

The ι rule differs slightly from the *ins* rule, it handles operations returning a computation Σ^*`A` instead of a value `A`. Thus it forwards the handling before combining the results.

```
ihandle (op_i (p: P_i,◇: N_i -> Σ*A)) with ih
= g_A (p, λn. ihandle (◇ n) with ih)
```

Example 4.4. The inductive handler for the state passing function examples is implemented as:

```
iStateMon = ihandler
| ϵ (a: A) -> λ(s:S). (s,a)
| get (_:(),k:S -> S -> (S,A)) -> λ(s:S). k s s
| put (p S,k:() -> S -> (S,A)) -> λ(s:S). k () p
```

4.3 Expressiveness of Monoidal and Inductive Handlers

Both the free monoid and initial algebra bases have an equal expressiveness. Each can present the interface of the other. There are also two properties to ensure the consistency between each basis, the round-trip and coherency properties. The former requires that a round-trip conversion, namely converting to one basis and then back to the other basis, is the identity. The latter requires that the handlers behave in a consistent manner in both bases.

Initial Algebra Basis from Free Monoid Basis. The following definitions represent the constructor/handler from the initial algebra basis:

$$\begin{aligned} \iota &= \Sigma \otimes \Sigma^* \xrightarrow{ins \otimes \Sigma^*} \Sigma^* \otimes \Sigma^* \xrightarrow{\mu} \Sigma^* \\ eval_X\, e &= X^X \xrightarrow{\rho^{-1}} X^X \otimes I \xrightarrow{X^X \otimes e} X^X \otimes X \xrightarrow{ev_X} X \\ (\!|[e,g]|\!) &= \Sigma^* \xrightarrow{free\, \lfloor g \rfloor} X^X \xrightarrow{eval_X\, e} X \end{aligned}$$

Where the use of *free* $\lfloor g \rfloor$ is justified, since it interprets to the endomorphism monoid $(X^X, \dot{e} : I \to X^X, \dot{m} : X^X \otimes X^X \to X^X)$.

Free Monoid Basis from Initial Algebra Basis. The following definitions represent the constructors/handler from the free monoid basis:

$$\begin{aligned} ins &= \Sigma \xrightarrow{\rho_\Sigma^{-1}} \Sigma \otimes I \xrightarrow{\Sigma \otimes \epsilon} \Sigma \otimes \Sigma^* \xrightarrow{\iota} \Sigma^* \\ \mu &= \lceil (\!|[\, \lfloor I \otimes \Sigma^* \xrightarrow{\lambda_{\Sigma^*}} \Sigma^* \rfloor, \\ &\quad \lfloor (\Sigma \otimes \Sigma^{*\Sigma^*}) \otimes \Sigma^* \xrightarrow{\alpha^{-1}} \Sigma \otimes (\Sigma^{*\Sigma^*} \otimes \Sigma^*) \\ &\quad \xrightarrow{\Sigma \otimes ev_{\Sigma^*}} \Sigma \otimes \Sigma^* \xrightarrow{\iota} \Sigma^* \rfloor\,]|\!) \rceil \\ free\, f &= (\!|[I \xrightarrow{e} M, \Sigma \otimes M \xrightarrow{(f \otimes M)} M \otimes M \xrightarrow{m} M]|\!) \end{aligned}$$

Where (M, e, m) is a monoid.

Properties. The round-trip properties are obtained by deriving the definition of the constructors, from the other basis, as a property. The proofs of these properties are in Appendix B.3, B.4, C.3, C.4 and C.5.

The coherency properties are obtained by deriving the evaluation rules of the handler, from the other basis, as a property. The proofs of these properties are in Appendix B.5 and C.6.

4.4 Expressiveness of Monoidal and Free Algebra Handlers

The monoidal handler for monads is slightly different from the original handlers based on free algebras. At first sight it seems that the carriers of the two handlers only coincide when the carrier B of the free algebra handler is of the form MA where M is the monad carrier of the monoidal handler and the free algebra handler is natural in A. This might suggest that the monoidal handler is less expressive than its free algebra counterpart, which is not restricted to this particular form of carrier. However, both handlers are equally expressive.

The continuation monad enables translating the `handle` interface in terms of `ihandle` or `mhandle`. The continuation monad is defined as X^{X^A}, which is the type `(A -> X) -> X` in $\mathscr{C}$. The translation of `handle` in terms of `ihandle` is:

```
handle x with
  (handler
   | val (a: A) -> ...: X                  (v)
   | op_i (p_i: P_i,k: N_i -> X) -> ...: X  (c_i)
```

Table 1: Overview of Handlers

	Free Algebra	Free Monoid (*free*)	Free Monoid ($(\!|-|\!)$)
syntax /computation	$\texttt{val}_A : A \to \Sigma^* A$ $\texttt{op}_A : \Sigma(\Sigma^* A) \to \Sigma^* A$	$\epsilon : I \to \Sigma^*$ $ins : \Sigma \to \Sigma^*$ $\mu : \Sigma^* \otimes \Sigma^* \to \Sigma^*$	$\epsilon : I \to \Sigma^*$ $\iota : \Sigma \otimes \Sigma^* \to \Sigma^*$
handler	Σ-algebra: $\langle B, c = [c_0, \ldots, c_n] : \Sigma B \to B \rangle$ $v : A \to B$	monoid: (M, e, m) $f = [f_0, \ldots, f_n] : \Sigma \to M$	$I + \Sigma \otimes -$-algebra: $\langle X, [e, g] : I + \Sigma \otimes X \to X \rangle$ $g = \lceil [\lfloor g_0 \rfloor, \ldots, \lfloor g_n \rfloor] \rceil$
handler (clauses)	`| val` A `->` B $\quad (v)$ `| op`$_i$ $\Sigma_i B$ `->` B $\quad (c_i)$	`|` ϵ I `->` M $\quad (e)$ `| op`$_i$ Σ_i `->` M $\quad (f_i)$ `|` μ $M \otimes M$ `->` M $\quad (m)$	`|` ϵ I `->` X $\quad (e)$ `| op`$_i$ $\Sigma_i \otimes X$ `->` X $\quad (g_i)$
handling a computation	$\texttt{handle} : \Sigma^* A \to B$	$free\ f : \Sigma^* \to M$	$(\!|[e, g]|\!) : \Sigma^* \to X$
handling rules	$\texttt{handle} \circ \texttt{val}_A = v$ $\texttt{handle} \circ \texttt{op}_A = c \circ \Sigma\texttt{handle}$	$free\ f \circ \epsilon = e$ $free\ f \circ ins = f$ $free\ f \circ \mu = m \circ (free\ f \otimes free\ f)$	$(\!|[e, g]|\!) \circ \epsilon = e$ $(\!|[e, g]|\!) \circ \iota = g \circ (\Sigma \otimes (\!|[e, g]|\!))$

```
  )
= (ihandle x with
    (ihandler
     | ϵ (a: A)
         -> λ(f: A -> X). f a
     | op_i (p_i: P_i,k: N_i -> ((A -> X) -> X))
         -> λ(f:A -> X). c_i (p_i, λ(n: N_i). k n f)
    )) v
```

Where the `ihandler` interprets to the `(A -> X) -> X` type and is then evaluated with v.

The consistency property of this translation is proven in Appendix D.1 and D.2.

4.5 Summary

An overview of the free algebra and free monoid approach can be seen in table 1. Since free monoid handlers are equivalent to free algebra handlers, when the former is instantiated for monads, we consider it a natural extension of the latter approach. By instantiating the free monoid handlers for other effects such as applicative functors or arrows it is possible to define handlers for non-monadic effects, which we call non-monadic handlers.

5 NON-MONADIC HANDLERS

This section explores the monoidal handlers for applicatives and arrows. We instantiate the monoidal categories accordingly and specialize the definitions of the derived handlers.

5.1 Applicative Handlers

Instantiating the syntax morphisms for $End(\mathscr{C})$ with Day convolution $\star$ gives:

$$\begin{aligned} \epsilon_A &: A \to \Sigma^* A \\ \iota_A &: (\Sigma \star \Sigma^*)(A) \to \Sigma^* A \\ &= \left(\int^Z \Sigma(Z \to A) \times \Sigma^* Z\right) \to \Sigma^* A \end{aligned}$$

Thus, $\iota_{Z,A} : \Sigma(Z \to A) \times \Sigma^* Z \to \Sigma^* A$. The parameter of type $\Sigma(Z \to A) = P_0 \times (Z \to A)^{N_0} + \ldots + P_n \times (Z \to A)^{N_n}$ denotes one of the possible operations, which is to be embedded into the computation. Each operation has a parameter of type P_i and a morphism of type $(Z \to A)^{N_i}$. This morphism combines the result N_i of the operation with the result Z of the rest of the computation. Parameter $\Sigma^* Z$ is the rest of the computation or the "continuation". Unlike the monadic continuation, $\Sigma^* Z$ does not depend on the result N_i of the preceding operation.

Here is an example applicative program:

```
exProgApp = do x <- get ()
   put 5
   return x
```

which desugars into the following primitive syntax:

```
ι_(),ℕ (get ((),λ(n:ℕ).λ(z:()).n)) (
  ι_(),() (put (5,λ(n:()).λ(z:()).())) (
    ϵ_() ()
  )
)
```

Example 5.1. Let us first implement a monoidal handler and its inductive counterpart for interpreting the `get`/`put` operations, in the usual way, into state passing functions `S -> S×A`, which is an applicative functor in addition to being a monad.

The monoidal handler is implemented as:

```
mStateApp = mhandler
| ϵ (a: A) -> λ(s:S). (s,a)
| μ (mza: S -> (S,Z -> A),mz: S -> (S,Z))
```

```
    -> λ(s:S) let (s':S,f:Z->A) = mza s
                  (s":S,z:Z) = mz s'
              in (s",f z)
| get (_: (),f: S -> A) -> λ(s:S). ((),f s)
| put (p: S,f: () -> A) -> λ(s:S). (p,f ())
```

While the inductive handler is implemented as:

```
iStateApp = ihandler
| ϵ (a: A) -> λ(s:S). (s,a)
| get (_: (),f: S -> Z -> A,k: S -> (S,Z))
    -> λ(s:S). let (s':S,z:Z) = k s
               in (s',f s z)
| put (p: S,k: () -> Z -> A,f: S -> (S,Z))
    -> λ(s:S). let (s':S,z:Z) = k s
               in (s',f () z)
```

To illustrate the additional possibility for analysis that applicative functors enable, a handler counting the number of `put 5` operations is implemented below. For that purpose it interprets programs in terms of the constant applicative functor $\Delta_{\mathbb{N}}\,\mathtt{A}$ and uses $\dot{+}$ to apply + on the natural numbers inside two $\Delta_{\mathbb{N}}\,\mathtt{A}$ values.
The monoidal handler implementation is:

```
mhandler
| ϵ (a: A) -> Δ_ℕ 0
| μ (mza: Δ_ℕ (Z -> A),mz: Δ_ℕ Z) -> mza ∔ mz
| get (_: (),f: ℕ -> A) -> Δ_ℕ 0
| put (p: ℕ,f: () -> A)
    -> if (p == 5) then Δ_ℕ 1 else Δ_ℕ 0
```

Where we interpret to the applicative $(\Delta_{\mathbb{N}}, \lambda\_.\Delta_{\mathbb{N}}\,0, \dot{+})$. The applicative laws are satisfied since $(\mathbb{N}, 0, +)$ is a monoid.

The inductive handler implementation is:

```
ihandler
| ϵ (a: A) -> Δ_ℕ 0
| get (_: (),f: ℕ -> Z -> A,k: Δ_ℕ Z) -> k
| put (p: ℕ,f: () -> Z -> A,k: Δ_ℕ Z)
    -> if (p == 5) then k ∔ 1 else k
```

5.2 Arrow Handlers

Instantiating the syntax morphisms for $SPro(\mathscr{C})$ with Profunctor composition $\otimes$ gives:

$$\begin{aligned}\epsilon_{A,B} &: B^A \to \vec{\Sigma}^*(A,B)\\ \iota_{A,B} &: (\vec{\Sigma} \otimes \vec{\Sigma}^*)(A,B) \to \vec{\Sigma}^*(A,B)\\ &= (\int^Z \vec{\Sigma}(A,Z) \times \vec{\Sigma}^*(Z,B)) \to \vec{\Sigma}^*(A,B)\end{aligned}$$

Where the notation $\vec{\Sigma}$ denotes a profunctor signature. Thus, $\iota_{Z,A,B} : \vec{\Sigma}(A,Z) \times \vec{\Sigma}^*(Z,B) \to \vec{\Sigma}^*(A,B)$. The output Z of the embedded operation $\vec{\Sigma}(A,Z)$ is linked to the input of the rest of the computation $\vec{\Sigma}^*(Z,B)$.

These constructors require an altered view on operations. We require operations which are profunctors instead of functors.

Operations as Profunctors. Signature functors defined as $\Sigma_i B = P_i \times B^{N_i}$ can be extended to profunctors $\vec{\Sigma}_i(A,B) = (P_i \times B^{N_i})^A$. Profunctor versions for `get` and `put` become: $\vec{\Sigma}_{\{\mathtt{get}\}}(A,B) = ((\) \times B^{\mathtt{S}})^A$ and $\vec{\Sigma}_{\{\mathtt{put}\}}(A,B) = (\mathtt{S} \times B^{(\)})^A$.

An example program from this syntax:

```
ι_S,A,S (get (λ(_:A).(),λ(s:S).s)) (
  ι_S,S,S (put (λ(a:S).a,λ(_:()).a)) (
    ϵ_S,S (λ(s:S).s)
  )
)
```

Which, in sugared notation, is equivalent to:

```
do x <- get ()
   put x
   return x
```

Example 5.2. We implement handlers interpreting `get`/`put` to a profunctor version of state passing functions `(A,S) -> (S,B)`.
The monoid handler is implemented as:

```
mStateArr = mhandler
| ϵ (f: A -> B) -> λ(a:A,s:S). (s,f a)
| μ (maz:(A,S) -> (S,Z),mzb:(Z,S) -> (S,B))
    -> λ(a:A,s:S). let (s':S,z:Z) = maz (a,s)
                       (s":S,b:B) = mzb (z,s')
                   in (s",b)
| get (p: A -> ((),S -> B))
    -> λ(a:A,s:S). let (_:(),k:S->B) = p a
                   in (s,k s)
| put (p: A -> (S,() -> B))
    -> λ(a:A,s:S). let (s':S,k:()->B) = p a
                   in (s',k ())
```

While the inductive handler is implemented as:

```
iStateArr = ihandler
| ϵ (f: A -> B) -> λ(a:A,s:S). (s,f a)
| get (p: A -> ((),S -> Z),k: (Z,S) -> (S,B))
    -> λ(a:A,s:S). let (_:(),f:S->Z) = p a
                   in k (f s,s)
| put (p: A -> (S,() -> Z),k: (Z,S) -> (S,B))
    -> λ(a:A,s:S). let (s':S,f:()->Z) = p a
                   in k (f (),s')
```

To illustrate the analysis possibilities of arrows, we show an arrow handler that counts `put` operations below. Expressing the counting of `put 5` operations is impossible for arrow handlers, since the arguments to operations are not statically known. In this case the handlers interpret to the constant profunctor $\vec{\Delta}_{\mathbb{N}}(\mathtt{A},\mathtt{B})$.
The monoid handler implementation is:

```
mhandler
| ϵ (f: A -> B) -> Δ⃗ℕ 0
| μ (maz: Δ⃗ℕ(A,Z),mzb: Δ⃗ℕ(Z,B)) -> maz + mzb
| get (p: A -> ((),S -> B)) -> Δ⃗ℕ 0
| put (p: A -> (S,() -> B)) -> Δ⃗ℕ 1
```

Where we interpret to the arrow $(\vec{\Delta}_{\mathbb{N}}, \lambda\_.\vec{\Delta}_{\mathbb{N}}\, 0, \dot{+})$. Again, the arrow laws are satisfied since $(\mathbb{N}, 0, +)$ is a monoid.
The inductive handler implementation is:

```
ihandler
| ϵ (f: A -> B) -> Δ⃗ℕ
| get (p: A -> ((),S -> Z),k: Δ⃗ℕ(Z,B)) -> k
| put (p: A -> (S,() -> Z),k: Δ⃗ℕ(Z,B)) -> k ∔ 1
```

6 REUSING HANDLERS AND PROGRAMS

In the previous section we have seen handlers for different computation classes, interpreting programs as state passing functions. There is no essential difference in how these handlers operate. Raising the question wether we can reuse handler definitions across the computation classes. Dually, programs from different computation classes may express the same computation. For example, an applicative computation can be identical to a monadic computation that does not use the full monadic expressiveness. Again, raising the question whether we can reuse computations across classes.

This section accomplishes both forms of reuse by means of an adjunction whose right adjoint is a lax monoidal functor. However, not every form of reuse is possible. We can only reuse programs from less expressive classes in more expressive classes, for example applicative → arrow → monad. The handler reuse opportunities are dual, that is monad → arrow → applicative.

6.1 Background

Before explaining how to reuse handlers and programs, we introduce the two key concepts.

Adjunction. The pair of functors $F : \mathscr{A} \to \mathscr{B}$ and $G : \mathscr{B} \to \mathscr{A}$ map between the two monoidal categories $\mathscr{A}$ and $\mathscr{B}$. These functors are related by the adjunction $G \dashv F$:

This adjunction is characterized by the isomorphism:

$$\llparenthesis - \rrparenthesis : \mathscr{A}(GA, B) \cong \mathscr{B}(A, FB) : \llbracket - \rrbracket$$

(Co-)Lax Monoidal Functors. Functors F and G have to preserve the monoidal structure of the two categories. In particular, the right adjoint F has to be a *lax monoidal functor* [10], signified by the double arrow in the adjunction diagram.

Definition 6.1 (Lax monoidal functor). Let $(\mathscr{A}, \otimes, I^{\mathscr{A}}, \alpha^{\mathscr{A}}, \lambda^{\mathscr{A}}, \rho^{\mathscr{A}})$ and $(\mathscr{B}, \oplus, I^{\mathscr{B}}, \alpha^{\mathscr{B}}, \lambda^{\mathscr{B}}, \rho^{\mathscr{B}})$ be two monoidal categories. A lax monoidal functor between them is

a) a functor $F : \mathscr{A} \to \mathscr{B}$
b) a morphism $\phi^o : I^{\mathscr{B}} \to F(I^{\mathscr{A}})$
c) a natural transformation $\phi_{A,B} : FA \oplus FB \to F(A \otimes B)$

satisfying coherence conditions with respect to unitality and associativity.

The key property of lax monoidal functors is their mapping of monoids (M, e, m) in $\mathscr{A}$ to monoids (FM, e', m') in $\mathscr{B}$, where e' and m' are defined as:

$$\begin{aligned} e' &= I^{\mathscr{B}} \xrightarrow{\phi^o} F(I^{\mathscr{A}}) \xrightarrow{F\,e} FM \\ m' &= FM \oplus FM \xrightarrow{\phi_{M,M}} F(M \otimes M) \xrightarrow{F\,m} FM \end{aligned}$$

Dually and as a consequence of the adjunction, the left adjoint G is a colax monoidal functor. This means it has morphisms $\psi^o : G(I^{\mathscr{B}}) \to I^{\mathscr{A}}$ and $\psi_{A,B} : G(A \oplus B) \to GA \otimes GB$ that satisfy similar conditions.

6.2 Transformation-based Approach

This section presents our approach, based on transforming the programs/handlers in one category to programs/handlers in the other category.

Notation. To prevent confusing the category of signatures and programs, we use the following notational convention. Signatures that originate in category $\mathscr{A}$ are denoted Σ, and free monoids are superscripted with $\mathscr{A}$: $\Sigma^{\mathscr{A}}$, rather than *. Signatures from category $\mathscr{B}$ are denoted Ξ and free monoids have a superscript $\mathscr{B}$: $\Xi^{\mathscr{B}}$.

Signatures Ξ and Σ need to be related, they need to refer to the same operations. This relation is established by a morphism $f : G\Xi \to \Sigma$, which through the adjunction is isomorphic to $g : \Xi \to F\Sigma : \llparenthesis f \rrparenthesis$.

Also, $\mathscr{A}$ is the category with less expressive handlers, but more expressive programs. The opposite is true for $\mathscr{B}$: it has more expressive handlers, but less expressive programs.

Algebra Conversion. We now show how to convert a handler algebra $[i : I^{\mathscr{A}} \to X, a : \Sigma \otimes X \to X]$ in $\mathscr{A}$ to a handler algebra $[i' : I^{\mathscr{B}} \to FX, a' : F\Sigma \oplus FX \to FX]$ in $\mathscr{B}$. We precompose the F-mapped algebra morphisms with ϕ^o and $\phi_{\Sigma,X}$ respectively to obtain the converted algebra.

$$\begin{aligned} i' &= I^{\mathscr{B}} \xrightarrow{\phi^o} F(I^{\mathscr{A}}) \xrightarrow{Fi} FX \\ a' &= F\Sigma \oplus FX \xrightarrow{\phi_{\Sigma,X}} F(\Sigma \otimes X) \xrightarrow{Fa} FX \end{aligned}$$

In other words, we obtain a $\mathscr{B}$-handler $h' : (F\Sigma)^{\mathscr{B}} \to FX = \llparenthesis [i', a'] \rrparenthesis$ from an $\mathscr{A}$-handler $h : \Sigma^{\mathscr{A}} \to X = \llparenthesis [i, a] \rrparenthesis$.

For example, using this algebra conversion we can convert the handler `iStateArr` from Example 5.2 to a handler equivalent to `iStateApp` from Example 5.1, allowing the handling of applicative computations with `iStateArr`.

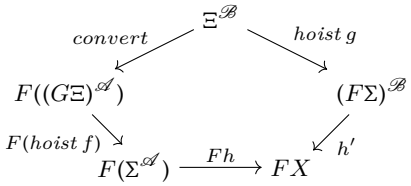
Program Conversion. The dual conversion, using the same elements, takes a program $\Xi^{\mathscr{B}}$ to a program $F((G\Xi)^{\mathscr{A}})$:

$$\begin{array}{rcl} [\![ins_{G\Xi}]\!] & : & \Xi \to F((G\Xi)^{\mathscr{A}}) \\ convert & = & \Xi^{\mathscr{B}} \xrightarrow{free\,[\![ins_{G\Xi}]\!]} F((G\Xi)^{\mathscr{A}}) \end{array}$$

Using *free i* requires that $F((G\Sigma)^{\mathscr{A}})$ induces a monoid, which it does since F is a lax monoidal functor.

For example, using this program conversion we can convert the program `exProgApp` represented with applicative syntax from Section 5.1 to a representation with the arrow syntax, allowing the use of arrow handlers on `exProgApp`.

Overview. The two conversions enable alternative paths for handling a program $\Xi^{\mathscr{B}}$ to a result FX,



, where *hoist* is defined as:

$$\begin{array}{rcl} x & : & A \to B \\ hoist\, x & : & A^* \to B^* = (\!|[\epsilon, \iota \circ (x \otimes B^*)]|\!) \end{array}$$

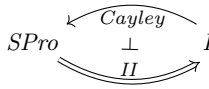
Appendix E proves that both paths of this diagram are equivalent to the fused morphism $(\!|[i', a' \circ (g \oplus FX)]|\!)$.

This means that converting a handler and handling a program, or converting this program and then handling it with the handler give the same result. The fused morphism is a likely optimization to the previous two, more intuitive, approaches.

6.3 Instances

This section instantiates the approach for three conversions: applicative $\leftrightarrow$ arrow, arrow $\leftrightarrow$ monad and applicative $\leftrightarrow$ monad. Instead of superscripts $^{\mathscr{A}}$ and $^{\mathscr{B}}$, each of applicative, arrow and monad have respectively $^{\mathtt{I}}$, $^{\rightsquigarrow}$ and $^{\mathtt{T}}$ as superscript.

Applicative $\leftrightarrow$ Arrow. For this conversion, we use the adjunction:

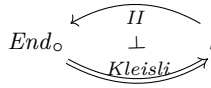


Here, the left adjoint G is $Cayley(F)(A,B) = F(B^A) : End_\star \to SPro$, which creates a (strong) profunctor by putting a contravariant argument inside the transformed functor.

The right adjoint functor F is $II(F)(A) = F((), A) : SPro \to End_\star$,[1] which transforms a profunctor into a functor by putting the unit value `()` in the contravariant position. It induces a lax monoidal functor $(II, \phi^{II} : II(F) \star II(G) \to II(F \otimes G), \phi^{o,II} : I^{\mathtt{I}} \to II(I^{\rightsquigarrow}))$.

This adjunction results in a handler algebra conversion: given $I^{\rightsquigarrow} \to X$ and $\Sigma \otimes X \to X$, it forms $I^{\mathtt{I}} \to II\,X$ and $II\,\Sigma \star II\,X \to II\,X$. It also results in a program conversion: $convert^{\mathtt{I}}_{\Xi} : \Xi^{\mathtt{I}} \to II((Cayley\,\Xi)^{\rightsquigarrow})$.

Arrow $\leftrightarrow$ Monad. For this conversion, we use the adjunction:

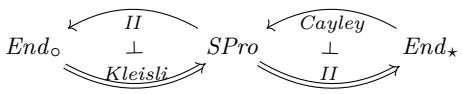


The left adjoint G is II from the previous paragraph.

The right adjoint functor F is $Kleisli(F)(A,B) = (FB)^A$, which creates a profunctor by putting a contravariant argument on the transformed functor. It induces a lax monoidal functor $(Kleisli, \phi^{Kl} : Kleisli(F) \otimes Kleisli(G) \to Kleisli(F \circ G), \phi^{o,Kl} : I^{\rightsquigarrow} \to Kleisli(I^{\mathtt{T}}))$, called Kleisli [14].

This adjunction results in a handler algebra conversion: given $I^{\mathtt{T}} \to X$ and $\Sigma \circ X \to X$, it forms $I^{\rightsquigarrow} \to II\,X$ and $Kleisli\,\Sigma \otimes Kleisli\,X \to Kleisli\,X$. It also results in a program conversion: $convert^{\rightsquigarrow}_{\Xi} : \Xi^{\rightsquigarrow} \to Kleisli((II\,\Xi)^{\mathtt{T}})$.

Applicative $\leftrightarrow$ Monad. We can stitch together the two previous adjunctions:



This results in the following algebra conversion:

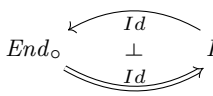
$$\begin{array}{rcl} i' & = & I^{\mathtt{I}} \xrightarrow{\phi^{o,II}} II(I^{\rightsquigarrow}) \xrightarrow{II(\phi^{o,Kl})} II(Kleisli\,I^{\mathtt{T}}) \\ & & \xrightarrow{II(Kleisli\,i)} II(Kleisli\,X) \\ a' & = & II(Kleisli\,\Sigma) \star II(Kleisli\,X) \xrightarrow{\phi^{II}} II(Kleisli\,\Sigma \otimes Kleisli\,X) \\ & & \xrightarrow{II(\phi^{Kl})} II(Kleisli(\Sigma \circ X)) \xrightarrow{II(Kleisli\,a)} II(Kleisli\,X) \end{array}$$

Given $i : I^{\mathtt{T}} \to X$ and $a : \Sigma \circ X \to X$.

It also results in the following program conversion:

$$\begin{array}{rcl} convert' & = & \Xi^{\mathtt{I}} \xrightarrow{convert^{\mathtt{I}}_{\Xi}} II((Cayley\,\Xi)^{\rightsquigarrow}) \\ & & \xrightarrow{II(convert^{\rightsquigarrow}_{Cayley\,\Xi})} II(Kleisli((II(Cayley\,\Xi))^{\mathtt{T}})) \end{array}$$

We can simplify the composition of the two adjunctions as both $II \circ Kleisli$ and $II \circ Cayley$ are isomorphic to $Id(F)(A) = FA$. The Id functor is adjoint to itself and induces the lax monoidal functor $(Id, \phi^{Day} : Id(F) \star Id(G) \to Id(F \circ G), \phi^{o,Day} : I^{\mathtt{I}} \to Id(I^{\mathtt{T}}))$, called Day [14].



This adjunction results in a handler algebra conversion: given $I^{\mathtt{T}} \to X$ and $\Sigma \circ X \to X$, it forms $I^{\rightsquigarrow} \to Id\,X$ and

[1] In the Haskell module `Control.Arrow` it is called `ArrowMonad`, https://hackage.haskell.org/package/base-4.10.0.0/docs/src/Control.Arrow.html#ArrowMonad.

$Id\,\Sigma \star Id\,X \to Id\,X$. It also results in a program conversion: $\Xi^{\mathtt{I}} \to Id((Id\,\Xi)^{\mathtt{T}})$.

7 RELATED WORK

Algebraic Effects and Handlers. This paper is an exploration in the space of interfaces which a language with algebraic effects and handlers could provide. The currently developed languages and libraries in this area (such as [1], [2], [5], [6], [8] and [13]) present the conventional monadic interface to the user, or only distinguish between applicative and monadic effects. Resulting in an interpretation limited to these effect classes.

The interest in abstractions for effects such as applicative and arrow motivates a broader handler interface, one which allows interpretations utilizing these, and potentially more, alternative abstractions. The motivation given at the start of the paper is a simple use case, but the overarching motivation is to port the use of these alternative abstractions to algebraic effects and handlers.

The methodology of derivation could be applied to other structures such as free near-semirings as explored by Rivas et al. [15]. Future work could explore the space of interfaces further to find a presentation which feels intuitive to a wide range of programmers.

Handlers for Idioms and Arrows. Lindley [7] introduces the calculus λ_{flow} which has handler constructs for monadic, applicative and arrow computations. The calculus has separate handling constructs for each of the different computation classes. We approach the same idea as a derivation from a general category theoretic framework. Lindley's and our interface slightly differ. The potential differences and similarities could be investigated further in future work. This work could also serve as a basis to give a denotational semantics for λ_{flow}.

8 CONCLUSION

This paper presents interfaces for applicative and arrow handlers derived from a unifying principle from which we also derive the conventional monadic handlers. This unifying principle are monoids in monoidal categories and was explored in detail by Rivas and Jaskelioff [14]. We show an equivalence between the initial algebra and free monoid syntax in the monoidal setting, as well as the initial algebra and free algebra approach in the monadic setting. We expand on the idea of lax monoidal functors with an adjunction to create a conversion of programs and handlers, enabling the reuse of handlers and programs across different monoidal categories.

ACKNOWLEDGEMENTS

This work was partially supported by project GRACeFUL, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 640954.

We also want to thank the reviewers for their feedback.

REFERENCES

[1] Andrej Bauer and Matija Pretnar. 2012. Programming with Algebraic Effects and Handlers. *CoRR* abs/1203.1539 (2012). http://arxiv.org/abs/1203.1539

[2] Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

[3] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and Loose Reasoning is Morally Correct. *SIGPLAN Not.* 41, 1 (Jan. 2006), 206–217. https://doi.org/10.1145/1111320.1111056

[4] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000).

[5] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (Aug. 2015), 94–105. https://doi.org/10.1145/2887747.2804319

[6] Daan Leijen. 2016. *Algebraic Effects for Functional Programming.* Technical Report. 15 pages. https://www.microsoft.com/en-us/research/publication/algebraic-effects-for-functional-programming/

[7] Sam Lindley. 2014. Algebraic Effects and Effect Handlers for Idioms and Arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2633628.2633636

[8] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

[9] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms Are Oblivious, Arrows Are Meticulous, Monads Are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229, 5 (March 2011), 97–117. https://doi.org/10.1016/j.entcs.2011.02.018

[10] Saunders Mac Lane. 1971. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag. Second edition, 1998.

[11] Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13.

[12] E. Moggi. 1989. *An abstract view of programming languages.* Technical Report ECS-LFCS-90-113. Edinburgh University, Department of Computer Science.

[13] Gordon Plotkin and Matija Pretnar. 2009. *Handlers of Algebraic Effects.* Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[14] Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of computation as monoids. *J. Funct. Program.* 27 (2017), e21. https://doi.org/10.1017/S0956796817000132

[15] Exequiel Rivas, Mauro Jaskelioff, and Tom Schrijvers. 2015. From Monoids to NearSemirings: The Essence of MonadPlus and Alternative. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP'15)*, Moreno Falaschi and Elvira Albert (Eds.). ACM, 196–207. https://doi.org/10.1145/2790449.2790514

[16] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 61–78.

A PROPERTIES *eval*

This section proves some auxiliary properties related to *eval*.

$$\begin{aligned} \dot{e} &= \lfloor \lambda_X \rfloor && (1)\\ \dot{m} &= \lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor && (2)\\ eval_X\, e &= ev_X \circ (X^X \otimes e) \circ \rho_{X^X}^{-1} && (3) \end{aligned}$$

$$eval_X\, a \circ \lfloor \lambda_X \rfloor = a \quad (4)$$

Proof.

$eval_X\, a \circ \lfloor \lambda_X \rfloor$
$=$ (def. $eval_X$ & def. ev_X)
$\lceil X^X \rceil \circ (X^X \otimes a) \circ \rho_{X^X}^{-1} \circ \lfloor \lambda_{X^X} \rfloor$
$=$ (ρ^{-1} is a natural transformation)
$\lceil X^X \rceil \circ (X^X \otimes a) \circ (\lfloor \lambda_X \rfloor \otimes I) \circ \rho_I^{-1}$
$=$ (bifunctor $\otimes$)
$\lceil X^X \rceil \circ (\lfloor \lambda_X \rfloor \otimes X) \circ (I \otimes a) \circ \rho_I^{-1}$
$=$ (naturality $\lceil - \rceil$)
$\lceil \lfloor \lambda_X \rfloor \rceil \circ (I \otimes a) \circ \rho_I^{-1}$
$=$ (inverses)
$\lambda_X \circ (I \otimes a) \circ \rho_I^{-1}$
$=$ (λ is a natural transformation)
$a \circ \lambda_I \circ \rho_I^{-1}$
$=$ (def. monoidal category)
$a \circ \rho_I \circ \rho_I^{-1}$
$=$ (inverses)
a □

$$\begin{aligned} a &: I \to X\\ b &: A \otimes X \to X\\ eval_X\, a \circ \lfloor b \rfloor &= b \circ (A \otimes a) \circ \rho_A^{-1} \end{aligned} \quad (5)$$

Proof.

$eval_X\, a \circ \lfloor b \rfloor$
$=$ (def. $eval_X$)
$\lceil X^X \rceil \circ (X^X \otimes a) \circ \rho_{X^X}^{-1} \circ \lfloor b \rfloor$
$=$ (ρ^{-1} is a natural transformation)
$\lceil X^X \rceil \circ (X^X \otimes a) \circ (\lfloor b \rfloor \otimes I) \circ \rho_A^{-1}$
$=$ (bifunctor $\otimes$)
$\lceil X^X \rceil \circ (\lfloor b \rfloor \otimes I) \circ (A \otimes a) \circ \rho_A^{-1}$
$=$ (naturality $\lceil - \rceil$)
$\lceil \lfloor b \rfloor \rceil \circ (A \otimes a) \circ \rho_A^{-1}$
$=$ (inverses)
$b \circ (A \otimes a) \circ \rho_A^{-1}$ □

B INITIAL ALGEBRA BASIS

This section proves the roundtrip and coherency properties for the initial algebra basis. first some relevant definitions are repeated, then each property related to a constructor or handler is proven in its own subsection.

B.1 Defining Properties $(\!|-|\!)$

$$\begin{aligned} (\!|[a,b]|\!) \circ \epsilon &= a && (6)\\ (\!|[a,b]|\!) \circ \iota &= b \circ (\Sigma \otimes (\!|[a,b]|\!)) && (7) \end{aligned}$$

B.2 Definition $\mu/ins/free$

$$\begin{aligned} \mu &= \lceil (\!|[\lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor]|\!) \rceil && (8)\\ ins &= \iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1} && (9)\\ free\, f &= (\!|[e, m \circ (f \otimes M)]|\!) && (10) \end{aligned}$$

B.3 Roundtrip Property ι

The roundtrip property is: $\iota = \mu \circ (ins \otimes \Sigma^*)$, the definition of ι in the free monoid basis.

We use the following local definitions to save some space:

$$\begin{aligned} b_1 &= \lfloor \lambda_{\Sigma^*} \rfloor\\ b_2 &= \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor\\ b &= [b_1, b_2] \end{aligned}$$

Proof.

$\mu \circ (ins \otimes \Sigma^*)$
$=$ (def. μ and ins)
$\lceil (\!|b|\!) \rceil \circ ((\iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1}) \otimes \Sigma^*)$
$=$ (naturality of $\lceil - \rceil$)
$\lceil (\!|b|\!) \circ \iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1} \rceil$
$=$ (property $(\!|-|\!)$ & bifunctor)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\!|b|\!) \circ \epsilon) \circ \rho_\Sigma^{-1} \rceil$
$=$ (property $(\!|-|\!)$)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes \lfloor \lambda_{\Sigma^*} \rfloor) \circ \rho_\Sigma^{-1} \rceil$
$=$ (naturality of $\lfloor - \rfloor$)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes \lfloor \lambda_{\Sigma^*} \rfloor) \otimes \Sigma^*) \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (α^{-1} is a natural transformation)
$\lceil \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ (\Sigma \otimes (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (bifunctor $\otimes$)
$\lceil \lfloor \iota \circ (\Sigma \otimes (ev_{\Sigma^*} \circ (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (definition ev_{Σ^*})
$\lceil \lfloor \iota \circ (\Sigma \otimes (\lceil \Sigma^{*^{\Sigma^*}} \rceil \circ (\lfloor \lambda_{\Sigma^*} \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (naturality of $\lceil - \rceil$)
$\lceil \lfloor \iota \circ (\Sigma \otimes \lceil \lfloor \lambda_{\Sigma^*} \rfloor \rceil) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (inverses)
$\lceil \lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ \rho_\Sigma^{-1} \rceil$
$=$ (naturality of $\lfloor - \rfloor$)
$\lceil \lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor \rceil$
$=$ (definition monoidal category, 3.2)
$\lceil \lfloor \iota \rfloor \rceil$
$=$ (inverses)
ι □

B.4 Roundtrip Property $(\!|-|\!)$

The roundtrip property is: $(\!|[e,g]|\!) = eval_X\, e \circ free \lfloor g \rfloor$, the definition of $(\!|-|\!)$ in the free monoid basis. We show that the

right-hand side is an algebra homomorphism $\Sigma^* \to X$. They are equal due to uniqueness of $(\!|[e, g]|\!)$.

PROOF.

$$
\begin{array}{ll}
& \mathit{eval}_X\ e \circ \mathit{free}\ \lfloor g \rfloor \circ \epsilon \\
= & \quad \text{(defs. } \mathit{free}\ \lfloor g \rfloor\text{)} \\
& \mathit{eval}_X\ e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!) \circ \epsilon \\
= & \quad \text{(property } (\!|-|\!)\text{)} \\
& \mathit{eval}_X\ e \circ \dot{e} \\
= & \quad \text{(def. } \dot{e}\text{)} \\
& \mathit{eval}_X\ e \circ \lfloor \lambda_X \rfloor \\
= & \quad \text{(property } \mathit{eval}_X\text{)} \\
& e
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{eval}_X\ e \circ \mathit{free}\ \lfloor g \rfloor \circ \iota \\
= & \quad \text{(def. } \mathit{free}\ \lfloor g \rfloor\text{)} \\
& \mathit{eval}_X\ e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!) \circ \iota \\
= & \quad \text{(property } (\!|-|\!)\text{)} \\
& \mathit{eval}_X\ e \circ \dot{m} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!)) \\
= & \quad \text{(introduce } \ldots \text{ to save some space)} \\
& \mathit{eval}_X\ e \circ \dot{m} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(def. } \dot{m}\text{)} \\
& \mathit{eval}_X\ e \circ \lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor \circ (\lfloor g \rfloor \otimes X^X) \\
& \quad \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(property } \mathit{eval}_X\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \circ ((X^X \otimes X^X) \otimes e) \\
& \quad \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots) \\
= & \quad (\alpha^{-1} \text{ is a natural transformation}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes e)) \circ \alpha^{-1} \\
& \quad \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor g \rfloor \otimes X^X) \circ (\Sigma \otimes \ldots) \\
= & \quad (\alpha^{-1} \text{ and } \rho^{-1} \text{ are natural transformations}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes e)) \\
& \quad \circ (\lfloor g \rfloor \otimes (X^X \otimes I)) \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(bifunctor } \otimes\text{)} \\
& \lceil X^X \rceil \circ (\lfloor g \rfloor \otimes X) \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \\
& \quad \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& g \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \circ \alpha^{-1} \circ \rho^{-1}_{\Sigma \otimes X^X} \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(property } \alpha^{-1} \circ \rho^{-1} = (id \otimes \rho^{-1})\text{)} \\
& g \circ (\Sigma \otimes \lceil X^X \rceil) \circ (\Sigma \otimes (X^X \otimes e)) \circ (\Sigma \otimes \rho^{-1}_{X^X}) \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(bifunctor } \otimes\text{)} \\
& g \circ (\Sigma \otimes (\lceil X^X \rceil \circ (X^X \otimes e) \circ \rho^{-1}_{X^X})) \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(def. } \mathit{eval}_X\ e\text{)} \\
& g \circ (\Sigma \otimes \mathit{eval}_X\ e) \circ (\Sigma \otimes \ldots) \\
= & \quad \text{(remove } \ldots \text{ \& bifunctor } \otimes\text{)} \\
& g \circ (\Sigma \otimes (\mathit{eval}_X\ e \circ (\!|[\dot{e}, \dot{m} \circ (\lfloor g \rfloor \otimes X^X)]|\!))) \\
= & \quad \text{(def. } \mathit{free}\ \lfloor g \rfloor\text{)} \\
& g \circ (\Sigma \otimes (\mathit{eval}_X\ e \circ \mathit{free}\ \lfloor g \rfloor)) \quad \square
\end{array}
$$

B.5 Coherency Properties *free f*

The *free f* morphism should have the same properties as in the free monoid basis, resulting in 3 coherency properties.

B.5.1 Property 1. We prove that $\mathit{free}\ f \circ \epsilon = e$.

PROOF.

$$
\begin{array}{ll}
& \mathit{free}\ f \circ \epsilon \\
= & \quad \text{(def. of } \mathit{free}\text{, 10)} \\
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \epsilon \\
= & \quad \text{(6)} \\
& e \quad \square
\end{array}
$$

B.5.2 Property 2. We prove that $\mathit{free}\ f \circ \mathit{ins} = f$.

PROOF.

$$
\begin{array}{ll}
& \mathit{free}\ f \circ \mathit{ins} \\
= & \quad \text{(defs. of } \mathit{ins} \text{ and } \mathit{free}\ f\text{)} \\
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(property of } (\!|-|\!)\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes (\!|[e, \ldots]|\!)) \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(bifunctor } \otimes\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes ((\!|[e, \ldots]|\!) \circ \epsilon)) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(property of } (\!|-|\!)\text{)} \\
& m \circ (f \otimes M) \circ (\Sigma \otimes e) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(bifunctor } \otimes\text{)} \\
& m \circ (M \otimes e) \circ (f \otimes I) \circ \rho_\Sigma^{-1} \\
= & \quad \text{(naturality of } \rho^{-1}\text{)} \\
& m \circ (M \otimes e) \circ \rho_M^{-1} \circ f \\
= & \quad \text{(monoid right unit property)} \\
& \rho_M \circ \rho_M^{-1} \circ f \\
= & \quad \text{(inverses)} \\
& f \quad \square
\end{array}
$$

B.5.3 Property 3. We prove that $\mathit{free}\ f \circ \mu = m \circ (\mathit{free}\ f \otimes \mathit{free}\ f)$. We first show that both sides are algebra homomorphisms $\Sigma^* \to M^{\Sigma^*}$, by uniqueness of $(\!|-|\!)$ both must be equal to $(\!|[\lfloor \mathit{free}\ f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]|\!)$.

PROOF. First we show that $\lfloor \mathit{free}\ f \circ \mu \rfloor = (\!|[\lfloor \mathit{free}\ f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]|\!)$

$$
\begin{array}{ll}
& \lfloor \mathit{free}\ f \circ \mu \rfloor \circ \epsilon \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor \mathit{free}\ f \circ \mu \circ (\epsilon \otimes \Sigma^*) \rfloor \\
= & \quad \text{(def. } \mu \text{ \& naturality of } \lceil - \rceil\text{)} \\
& \lfloor \mathit{free}\ f \circ \lceil (\!|b|\!) \circ \epsilon \rceil \rfloor \\
= & \quad \text{(def. } \mathit{free}\ f \text{ \& property of } (\!|-|\!)\text{)} \\
& \lfloor \mathit{free}\ f \circ \lambda_{\Sigma^*} \rfloor
\end{array}
$$

$$
\begin{array}{ll}
& \lfloor \mathit{free}\ f \circ \mu \rfloor \circ \iota \\
= & \quad \text{(naturality } \lfloor - \rfloor\text{)} \\
& \lfloor \mathit{free}\ f \circ \mu \circ (\iota \otimes \Sigma^*) \rfloor \\
= & \quad \text{(def. } \mu \text{ \& naturality of } \lceil - \rceil\text{)} \\
& \lfloor \mathit{free}\ f \circ \lceil (\!|b|\!) \circ \iota \rceil \rfloor \\
= & \quad \text{(property of } (\!|-|\!)\text{)}
\end{array}
$$

$\lfloor \mathit{free}\, f \circ \lceil b_2 \circ (\Sigma \otimes (\!|b|\!)) \rceil \rfloor$
$=$ (naturality of $\lceil - \rceil$)
$\lfloor \mathit{free}\, f \circ \lceil b_2 \rceil \circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (def. b_2 & inverses)
$\lfloor \mathit{free}\, f \circ \iota \circ (\Sigma \otimes \mathit{ev}_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (def. *free* f & property of $(\!|-|\!)$ & bifunctor $\otimes$)
$\lfloor m \circ (f \otimes (\mathit{free}\, f \circ \mathit{ev}_{\Sigma^*})) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (def. ev_{Σ^*} & naturality $\lceil - \rceil$)
$\lfloor m \circ (f \otimes \lceil \mathit{free}\, f^{\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (naturality of $\lceil - \rceil$)
$\lfloor m \circ (f \otimes (\lceil M^{\Sigma^*} \rceil \circ (\mathit{free}\, f^{\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1}$
$\circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (bifunctor $\otimes$)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1}$
$\circ ((\Sigma \otimes (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (naturality of α^{-1} & bifunctor)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ (\!|b|\!)) \otimes \Sigma^*) \rfloor$
$=$ (naturality of $\lfloor - \rfloor$)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ (\!|b|\!)))$
$=$ ($(\!|b|\!) = \lfloor \mu \rfloor$)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\mathit{free}\, f^{\Sigma^*} \circ \lfloor \mu \rfloor))$
$=$ (naturality of $\lfloor - \rfloor$)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes (\lfloor \mathit{free}\, f \circ \mu \rfloor))$

Then we show that $\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor = (\!|[\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]\!|)$

$\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \circ \epsilon$
$=$ (naturality of $\lfloor - \rfloor$)
$\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \circ (\epsilon \otimes A^*) \rfloor$
$=$ (bifunctor $\otimes$)
$\lfloor m \circ ((\mathit{free}\, f \circ \epsilon) \otimes \mathit{free}\, f) \rfloor$
$=$ (def. *free* f & property of $(\!|-|\!)$)
$\lfloor m \circ (e \otimes \mathit{free}\, f) \rfloor$
$=$ (bifunctor $\otimes$ & monoid left unit property)
$\lfloor \lambda_M \circ (I \otimes \mathit{free}\, f) \rfloor$
$=$ (naturality of λ)
$\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor$

$\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \circ \iota$
$=$ (naturality $\lfloor - \rfloor$ & bifunctor $\otimes$)
$\lfloor m \circ ((\mathit{free}\, f \circ \iota) \otimes \mathit{free}\, f) \rfloor$
$=$ (def. *free* f & property of $(\!|-|\!)$ & bifunctor $\otimes$)
$\lfloor m \circ (m \otimes M) \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f) \rfloor$
$=$ (monoid associativity property)
$\lfloor m \circ (M \otimes m) \circ \alpha^{-1} \circ ((f \otimes \mathit{free}\, f) \otimes \mathit{free}\, f) \rfloor$
$=$ (naturality of α^{-1})
$\lfloor m \circ (M \otimes m) \circ (f \otimes (\mathit{free}\, f \otimes \mathit{free}\, f)) \circ \alpha^{-1} \rfloor$
$=$ (bifunctor $\otimes$)
$\lfloor m \circ (f \otimes (m \circ (\mathit{free}\, f \otimes \mathit{free}\, f))) \circ \alpha^{-1} \rfloor$
$=$ (inverses)
$\lfloor m \circ (f \otimes \lceil \lfloor m_M \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \rceil) \circ \alpha^{-1} \rfloor$
$=$ (naturality of $\lceil - \rceil$)
$\lfloor m \circ (f \otimes (\lceil M^{\Sigma^*} \rceil \circ (\lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \otimes \Sigma^*)))$
$\circ \alpha^{-1} \rfloor$
$=$ (bifunctor $\otimes$ & naturality of α^{-1})
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1}$
$\circ ((\Sigma \otimes \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor) \otimes \Sigma^*) \rfloor$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor)$

Then, using $\lfloor \mathit{free}\, f \circ \mu \rfloor = (\!|[\lfloor \mathit{free}\, f \circ \lambda_{\Sigma^*} \rfloor, \lfloor m \circ (f \otimes \lceil M^{\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor]\!|) = \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor$, we show that the property holds.

$\mathit{free}\, f \circ \mu$
$=$ (inverses)
$\lceil \lfloor \mathit{free}\, f \circ \mu \rfloor \rceil$
$=$ (proven above)
$\lceil \lfloor m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \rfloor \rceil$
$=$ (inverses)
$m \circ (\mathit{free}\, f \otimes \mathit{free}\, f)$ □

C FREE MONOID BASIS

This section proves the roundtrip and coherency properties for the free monoid basis. first some relevant definitions are repeated, then each property related to a constructor or handler is proven in its own subsection.

C.1 Defining Properties *free*

$$\mathit{free}\, f \circ \epsilon = e \tag{11}$$
$$\mathit{free}\, f \circ \mathit{ins} = f \tag{12}$$
$$\mathit{free}\, f \circ \mu = m \circ (\mathit{free}\, f \otimes \mathit{free}\, f) \tag{13}$$

, where (M, e, m) is a monoid.

C.2 Definition $\iota / (\!|[e, g]|\!)$

$$\iota = \mu \circ (\mathit{ins} \otimes \Sigma^*) \tag{14}$$
$$(\!|[e, g]|\!) = \mathit{eval}_X\, e \circ \mathit{free} \lfloor g \rfloor \tag{15}$$

C.3 Roundtrip Property *ins*

The roundtrip property is: $\mathit{ins} = \iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1}$, the definition of *ins* in the initial algebra basis.

Proof.
$\iota \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1}$
$=$ (def ι, 14)
$\mu \circ (\mathit{ins} \otimes \Sigma^*) \circ (\Sigma \otimes \epsilon) \circ \rho_\Sigma^{-1}$
$=$ (bifunctor $\otimes$)
$\mu \circ (\Sigma^* \otimes \epsilon) \circ (\mathit{ins} \otimes I) \circ \rho_\Sigma^{-1}$

$=$ (monoid property)
$\rho_{\Sigma^*} \circ (ins \otimes I) \circ \rho_\Sigma^{-1}$
$=$ (ρ is a natural transformation)
$ins \circ \rho_\Sigma \circ \rho_\Sigma^{-1}$
$=$ (inverses)
ins $\square$

C.4 Roundtrip Property μ

The roundtrip property is: $\mu = \lceil (\!| \lfloor [\lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor] |\!) \rceil$, the definition of μ in the initial algebra basis. We prove this by using the fact that both sides (after $\lfloor - \rfloor$) are equal to $free\ \lfloor \iota \rfloor$.

C.4.1 Left-Hand Side. First we show that $\lfloor \mu \rfloor = free\ \lfloor \iota \rfloor$. We show that it is a monoid homomorphism $\Sigma^* \to \Sigma^{*\Sigma^*}$ and that $\lfloor \mu \rfloor \circ ins = \lfloor \iota \rfloor$. They are equal due to uniqueness of *free*.

Proof.
$\lfloor \mu \rfloor \circ \epsilon$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor \mu \circ (\epsilon \otimes \Sigma^*) \rfloor$
$=$ (monoid property)
$\lfloor \lambda_{\Sigma^*} \rfloor$
$=$ (def. $\dot{e}$)
$\dot{e}$

$\lfloor \mu \rfloor \circ ins$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor \mu \circ (ins \otimes \Sigma^*) \rfloor$
$=$ (def ι)
$\lfloor \iota \rfloor$

$\lfloor \mu \rfloor \circ \mu$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor \mu \circ (\mu \otimes \Sigma^*) \rfloor$
$=$ (monoid property)
$\lfloor \mu \circ (\Sigma^* \otimes \mu) \circ \alpha^{-1} \rfloor$
$=$ (inverses)
$\lfloor \lceil \lfloor \mu \rfloor \rceil \circ (\Sigma^* \otimes \lceil \lfloor \mu \rfloor \rceil) \circ \alpha^{-1} \rfloor$
$=$ (naturality $\lceil - \rceil$ & bifunctor $\otimes$)
$\lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \mu \rfloor \otimes (\lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \mu \rfloor \otimes \Sigma^*))) \circ \alpha^{-1} \rfloor$
$=$ (bifunctor $\otimes$)
$\lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ (\lfloor \mu \rfloor \otimes (\lfloor \mu \rfloor \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor$
$=$ (α^{-1} is a natural transformation)
$\lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ \alpha^{-1} \circ ((\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor) \otimes \Sigma^*) \rfloor$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\Sigma^{*\Sigma^*} \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ \alpha^{-1} \rfloor \circ (\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor)$
$=$ (def. $\dot{m}$)
$\dot{m} \circ (\lfloor \mu \rfloor \otimes \lfloor \mu \rfloor)$ $\square$

C.4.2 Right-Hand Side. We use the following local definitions for readability

$$\begin{aligned} b_1 &= \lfloor \lambda_{\Sigma^*} \rfloor \\ b_2 &= \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \end{aligned}$$

We show that $(\!| [b_1, b_2] |\!) = free\ \lfloor \iota \rfloor$. We show that it is a monoid homomorphism $\Sigma^* \to \Sigma^{*\Sigma^*}$ and that $(\!| [b_1, b_2] |\!) \circ ins = \lfloor \iota \rfloor$. They are equal due to uniqueness of *free*.

Proof.
$(\!| [b_1, b_2] |\!) \circ \epsilon$
$=$ (def. $(\!| - |\!)$)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ free\ \lfloor b_2 \rfloor \circ \epsilon$
$=$ (property *free*)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ \lfloor \lambda_{\Sigma^{*\Sigma^*}} \rfloor$
$=$ (property *eval*)
b_1
$=$ (expand b_1)
$\lfloor \lambda_{\Sigma^*} \rfloor$
$=$ (def. $\dot{e}$)
$\dot{e}$ $\square$

$(\!| [b_1, b_2] |\!) \circ ins$
$=$ (def. $(\!| - |\!)$)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ free\ \lfloor b_2 \rfloor \circ ins$
$=$ (property *free*)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ \lfloor b_2 \rfloor$
$=$ (property *eval*)
$b_2 \circ (\Sigma \otimes b_1) \circ \rho_\Sigma^{-1}$
$=$ (expand b_2)
$\lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\Sigma \otimes b_1) \circ \rho_\Sigma^{-1}$
$=$ (naturality $\lfloor - \rfloor$)
$\lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \circ ((\Sigma \otimes b_1) \otimes \Sigma^*) \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor$
$=$ (α^{-1} is a natural transformation)
$\lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ (\Sigma \otimes (b_1 \otimes \Sigma^*)) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor$
$=$ (def. *ev*)
$\lfloor \iota \circ (\Sigma \otimes \lceil \Sigma^{*\Sigma^*} \rceil) \circ (\Sigma \otimes (b_1 \otimes \Sigma^*)) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor$
$=$ (bifunctor $\otimes$ & naturality $\lceil - \rceil$)
$\lfloor \iota \circ (\Sigma \otimes \lceil b_1 \rceil) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor$
$=$ (expand b_1 & inverses)
$\lfloor \iota \circ (\Sigma \otimes \lambda_{\Sigma^*}) \circ \alpha^{-1} \circ (\rho_\Sigma^{-1} \otimes \Sigma^*) \rfloor$
$=$ (def. monoidal category)
$\lfloor \iota \rfloor$

$(\!| [b_1, b_2] |\!) \circ \mu$
$=$ (def. $(\!| - |\!)$)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ free\ \lfloor b_2 \rfloor \circ \mu$
$=$ ($free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor$)
$eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ \mu$
$=$ ($eval_{\Sigma^{*\Sigma^*}}\ b_1 \circ \lfloor \dot{m} \rfloor = \Sigma^{*\Sigma^*}$)
$free\ \lfloor \iota \rfloor \circ \mu$
$=$ (property *free*)
$\dot{m} \circ (free\ \lfloor \iota \rfloor \otimes free\ \lfloor \iota \rfloor)$

$$
\begin{array}{ll}
= & (eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor = \Sigma^{*\Sigma^*}) \\
& \dot{m} \circ ((eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor) \otimes \\
& \quad (eval_{\Sigma^*\Sigma^*}\ b_1 \circ \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor)) \\
= & (free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor) \\
& \dot{m} \circ ((eval_{\Sigma^*\Sigma^*}\ b_1 \circ free\ \lfloor b_2 \rfloor) \otimes (eval_{\Sigma^*\Sigma^*}\ b_1 \circ free\ \lfloor b_2 \rfloor)) \\
= & (\text{def. } (\!|-|\!)) \\
& \dot{m} \circ ((\!|[b_1, b_2]|\!) \otimes (\!|[b_1, b_2]|\!))
\end{array}
$$

Equality $free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor$ holds since

$$
\begin{array}{ll}
& \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ ins \\
= & (\text{property } free) \\
& \lfloor \dot{m} \rfloor \circ \lfloor \iota \rfloor \\
= & (\text{def. } \dot{m}) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \circ \lfloor \iota \rfloor \\
= & (\text{naturality } \lfloor - \rfloor) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \circ (\lfloor \iota \rfloor \otimes \Sigma^{*\Sigma^*}) \rfloor \\
= & (\text{naturality } \lfloor - \rfloor) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \circ ((\lfloor \iota \rfloor \otimes \Sigma^{*\Sigma^*}) \otimes \Sigma^*) \rfloor \rfloor \\
= & (\alpha^{-1} \text{ is a natural transformation}) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\Sigma^{*\Sigma^*} \otimes ev_{\Sigma^*}) \circ (\lfloor \iota \rfloor \otimes (\Sigma^{*\Sigma^*} \otimes \Sigma^*)) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{bifunctor } \otimes) \\
& \lfloor \lfloor ev_{\Sigma^*} \circ (\lfloor \iota \rfloor \otimes \Sigma^*) \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{def. } ev_{\Sigma^*}) \\
& \lfloor \lfloor \lceil \Sigma^{*\Sigma^*} \rceil \circ (\lfloor \iota \rfloor \otimes \Sigma^*) \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{naturality } \lceil - \rceil) \\
& \lfloor \lfloor \lceil \lfloor \iota \rfloor \rceil \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{inverses}) \\
& \lfloor \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor \rfloor \\
= & (\text{def. } b_2) \\
& \lfloor b_2 \rfloor
\end{array}
$$

, and $\lfloor \dot{m} \rfloor$ and $free\ \lfloor \iota \rfloor$ are both monoid homorphisms and thus their composition is a monoid homomorphism, meaning $\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ \epsilon = \ddot{e}$ and $\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \circ \mu = \ddot{m} \circ (\lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor \otimes \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor)$. Since $free\ \lfloor b_2 \rfloor$ is the unique monoid homomorphism, $free\ \lfloor b_2 \rfloor = \lfloor \dot{m} \rfloor \circ free\ \lfloor \iota \rfloor$.

Where $\dot{m}$ and $\ddot{m}$ are specialized to $\ddot{m} : (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})} \otimes (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})} \to (\Sigma^{*\Sigma^*})^{(\Sigma^{*\Sigma^*})}$ and $\dot{m} : \Sigma^{*\Sigma^*} \otimes \Sigma^{*\Sigma^*} \to \Sigma^{*\Sigma^*}$ for this case.

C.4.3 Property Proof. Then, using $(\!|[b_1, b_2]|\!) = free\ \lfloor \iota \rfloor = \lfloor \mu \rfloor$, we show that the roundtrip property holds.

Proof.

$$
\begin{array}{ll}
& \lceil (\!|[\lfloor \lambda_{\Sigma^*} \rfloor, \lfloor \iota \circ (\Sigma \otimes ev_{\Sigma^*}) \circ \alpha^{-1} \rfloor]|\!) \rceil \\
= & (\text{proven above}) \\
& \lceil \lfloor \mu \rfloor \rceil \\
= & (\text{inverses}) \\
& \mu \quad \square
\end{array}
$$

C.5 Roundtrip Property *free*

The roundtrip property is: $free\ f = (\!|[e, m \circ (f \otimes M)]|\!)$, the definition of $free\ f$ in the initial algebra basis. We show that the right-hand side is a monoid homomorphism $\Sigma^* \to M$ and $(\!|[e, m \circ (f \otimes M)]|\!) \circ ins = f$. They are equal due to uniqueness of *free*.

Proof.

$$
\begin{array}{ll}
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \epsilon \\
= & (\text{def. } (\!|-|\!)) \\
& eval_M\ e \circ free\ \lfloor m \circ (f \otimes M) \rfloor \circ \epsilon \\
= & (\text{property } free) \\
& eval_M\ e \circ \dot{e} \\
= & (\text{def. } \dot{e} \ \&\ \text{property } eval) \\
& e
\end{array}
$$

$$
\begin{array}{ll}
& (\!|[e, m \circ (f \otimes M)]|\!) \circ ins \\
= & (\text{def. } (\!|-|\!)) \\
& eval_M\ e \circ free\ \lfloor m \circ (f \otimes M) \rfloor \circ ins \\
= & (\text{property } free) \\
& eval_M\ e \circ \lfloor m \circ (f \otimes M) \rfloor \\
= & (\text{property } eval) \\
& m \circ (f \otimes M) \circ (\Sigma \otimes e) \circ \rho_\Sigma^{-1} \\
= & (\text{bifunctor } \otimes) \\
& m \circ (M \otimes e) \circ (f \otimes I) \circ \rho_\Sigma^{-1} \\
= & (\text{monoid property}) \\
& \rho_M \circ (f \otimes I) \circ \rho_\Sigma^{-1} \\
= & (\rho \text{ is a natural transformation}) \\
& f \circ \rho_\Sigma \circ \rho_\Sigma^{-1} \\
= & (\text{inverses}) \\
& f
\end{array}
$$

$$
\begin{array}{ll}
& (\!|[e, m \circ (f \otimes M)]|\!) \circ \mu \\
= & (\text{def. } (\!|-|\!)) \\
& eval_M\ e \circ free\ \lfloor m \circ (f \otimes M) \rfloor \circ \mu \\
= & (free\ \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\ f) \\
& eval_M\ e \circ \lfloor m \rfloor \circ free\ f \circ \mu \\
= & (eval_M\ e \circ \lfloor m \rfloor = M) \\
& free\ f \circ \mu \\
= & (\text{property } free) \\
& m \circ (free\ f \otimes free\ f) \\
= & (eval_M\ e \circ \lfloor m \rfloor = M) \\
& m \circ ((eval_M\ e \circ \lfloor m \rfloor \circ free\ f) \otimes (eval_M\ e \circ \lfloor m \rfloor \circ free\ f)) \\
= & (free\ \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\ f) \\
& m \circ ((eval_M\ e \circ free\ \lfloor m \circ (f \otimes M) \rfloor) \otimes (eval_M\ e \circ free\ \lfloor m \circ (f \otimes M) \rfloor)) \\
= & (\text{def. } (\!|-|\!)) \\
& m \circ ((\!|[e, m \circ (f \otimes M)]|\!) \otimes (\!|[e, m \circ (f \otimes M)]|\!))
\end{array}
$$

Equality $free\ \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ free\ f$ holds since

$$
\begin{array}{ll}
& \lfloor m \rfloor \circ free\ f \circ ins \\
= & (\text{property } free) \\
& \lfloor m \rfloor \circ f \\
= & (\text{naturality } \lfloor - \rfloor) \\
& \lfloor m \circ (f \otimes M) \rfloor
\end{array}
$$

, and $\lfloor m \rfloor$ and $\mathit{free}\, f$ are both monoid homorphisms and thus their composition is a monoid homomorphism, meaning $\lfloor m \rfloor \circ \mathit{free}\, f \circ \epsilon = \dot{e}$ and $\lfloor m \rfloor \circ \mathit{free}\, f \circ \mu = \dot{m} \circ (\lfloor m \rfloor \circ \mathit{free}\, f \otimes \lfloor m \rfloor \circ \mathit{free}\, f)$. Since $\mathit{free} \lfloor m \circ (f \otimes M) \rfloor$ is the unique monoid homomorphism, $\mathit{free} \lfloor m \circ (f \otimes M) \rfloor = \lfloor m \rfloor \circ \mathit{free}\, f$.

□

C.6 Coherency Properties $(\!|-|\!)$

The $(\!|[a,b]|\!)$ morphism should have the same properties as in the initial algebra basis, resulting in 2 coherency properties.

C.6.1 Property 1. We prove that $(\!|[a,b]|\!) \circ \epsilon = a$.

Proof.

$$
\begin{aligned}
& (\!|[a,b]|\!) \circ \epsilon \\
= \quad & \text{(def. } (\!|[a,b]|\!)\text{)} \\
& \mathit{eval}_X\, a \circ \mathit{free} \lfloor b \rfloor \circ \epsilon \\
= \quad & \text{(property } \mathit{free}\text{)} \\
& \mathit{eval}_X\, a \circ \lfloor \lambda_X \rfloor \\
= \quad & \text{(property } \mathit{eval}\text{)} \\
& a \quad \square
\end{aligned}
$$

C.6.2 Property 2. We prove that $(\!|[a,b]|\!) \circ \iota = b \circ (\Sigma \otimes (\!|[a,b]|\!))$.

Proof.

$$
\begin{aligned}
& (\!|[a,b]|\!) \circ \iota \\
= \quad & \text{(def. of } \iota \text{ and } (\!|[a,b]|\!)\text{)} \\
& \mathit{eval}_X\, a \circ \mathit{free} \lfloor b \rfloor \circ \mu \circ (\mathit{ins} \otimes \Sigma^*) \\
= \quad & \text{(property } \mathit{free}\text{)} \\
& \mathit{eval}_X\, a \circ \dot{m} \circ (\mathit{free} \lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \circ (\mathit{ins} \otimes \Sigma^*) \\
= \quad & \text{(bifunctor } \otimes \text{ \& } \mathit{free}\ f \circ \mathit{ins} = f\text{)} \\
& \mathit{eval}_X\, a \circ \dot{m} \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(def. } \mathit{eval}_X\, a\text{)} \\
& \mathit{ev}_X \circ (X^X \otimes a) \circ \rho^{-1}_{X^X} \circ \dot{m} \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & (\rho^{-1} \text{ natural transformation}) \\
& \mathit{ev}_X \circ (X^X \otimes a) \circ (\dot{m} \otimes I) \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(bifunctor } \otimes\text{)} \\
& \mathit{ev}_X \circ (\dot{m} \otimes X) \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(def. } \mathit{ev}_X \text{ and } \dot{m}\text{)} \\
& \lceil X^X \rceil \circ (\lfloor \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ \alpha^{-1} \rfloor \otimes X) \\
& \quad \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& \lceil X^X \rceil \circ (X^X \otimes \mathit{ev}_X) \circ \alpha^{-1} \circ ((X^X \otimes X^X) \otimes a) \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & (\alpha^{-1} \text{ natural transformation}) \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes a)) \circ \alpha^{-1} \circ \rho^{-1}_{X^X \otimes X^X} \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(property } \alpha^{-1} \circ \rho^{-1} = id \otimes \rho^{-1}\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes \lceil X^X \rceil) \circ (X^X \otimes (X^X \otimes a)) \circ (X^X \otimes \rho^{-1}_{X^X}) \\
& \quad \circ (\lfloor b \rfloor \otimes \mathit{free} \lfloor b \rfloor) \\
= \quad & \text{(bifunctor } \otimes\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes (\lceil X^X \rceil \circ (X^X \otimes a) \circ \rho^{-1}_{X^X} \circ \mathit{free} \lfloor b \rfloor)) \\
= \quad & \text{(def. of } \mathit{eval}_X\, a\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes (\mathit{eval}_X\, a \circ \mathit{free} \lfloor b \rfloor)) \circ (\lfloor b \rfloor \otimes \Sigma^*) \\
= \quad & \text{(def. of } (\!|[a,b]|\!)\text{)} \\
& \lceil X^X \rceil \circ (X^X \otimes (\!|[a,b]|\!)) \circ (\lfloor b \rfloor \otimes \Sigma^*) \\
= \quad & \text{(bifunctor } \otimes\text{)} \\
& \lceil X^X \rceil \circ (\lfloor b \rfloor \otimes X^X) \circ (\Sigma \otimes (\!|[a,b]|\!)) \\
= \quad & \text{(naturality } \lceil - \rceil \text{ \& inverses)} \\
& b \circ (\Sigma \otimes (\!|[a,b]|\!)) \quad \square
\end{aligned}
$$

D COHERENCY PROPERTIES `HANDLE`

The `handle _ with h` operation in the initial algebra basis should have the same properties as in the free algebra basis. This results in 2 coherency properties for the operation and value rule respectively.

In the following proofs, we assume the following is defined:

```
h =
  handler
  | val (a: A) -> ...: X                                   (v)
  | op_i (p_i: P_i,k: N_i -> X) -> ...: X                  (c_i)
ih =
   ihandler
   | ϵ (a: A)
       -> λ(f: A -> X). f a                                (e)
   | op_i (p_i: P_i,k: N_i -> ((A -> X) -> X))
       -> λ(f:A -> X). c_i (p_i, λ(n: N_i). k n f)(g_i)
```

D.1 Coherency Property: Value Rule

```
  handle (x: A) with h
= (def. handle)
  (ihandle (x: A) with ih) v
= (ϵ rule)
  (e x) v
= (def. e)
  ((λ a. λf. f a) x) v
= (application)
  (λf. f x) v
= (application)
  v x
```

D.2 Coherency Property: Operation Rule

```
  handle (op_i (p: P_i, ◇: N_i -> Σ*A)) with h
= (def. handle)
  (ihandle (op_i (p: P_i, ◇: N_i -> Σ*A)) with ih) v
= (ι rule)
  (g_i (p, λn. ihandle (◇ n) with ih)) v
= (def. g_i)
  ((λ(p_i,k). λf. c_i (p_i, λn. k n f))
      (p, λn. ihandle (◇ n) with ih)) v
= (α-renaming & application)
  (λf. c_i (p, λn. (λx. ihandle (◇ x) with ih) n f)) v
= (application)
```

c_i `(p, (λn. λx. ihandle (◇ x) with ih) n` v`))`
`= (application)`
c_i `(p, (λn. ihandle (◇ n) with ih)` v`)`
`= (def. handle)`
c_i `(p, λn. handle (◇ n) with h)`

E CONVERSION DIAGRAM

We show that both paths of the diagram are equal to $(\!|[i', a' \circ (g \oplus FX)]|\!)$. First we repeat some relevant definitions, then prove the algebra conversion path and lastly prove the program conversion path.

E.1 Definitions

E.1.1 Signature Conversion.

$$\begin{aligned} f &: G\Xi \to \Sigma \\ g &: \Xi \to F\Sigma = \llbracket f \rrbracket \end{aligned}$$

E.1.2 Algebra Conversion. The original algebra components are:

$$\begin{aligned} i &: I^{\mathscr{A}} \to X \\ a &: \Sigma \otimes X \to X \end{aligned}$$

The transformed algebra components are:

$$\begin{aligned} i' &: I^{\mathscr{B}} \to FX = Fi \circ \phi^o \\ a' &: F\Sigma \oplus FX \to FX = Fa \circ \phi \end{aligned}$$

E.2 Algebra Conversion

We have to show that $h' \circ \mathit{hoist}\, g = (\!|[i', a' \circ (g \oplus FX)]|\!)$.

Proof. Both $h' = (\!|[i', a']|\!)$ and $\mathit{hoist}\, g = (\!|[\epsilon, \iota \circ (g \oplus (F\Sigma)^{\mathscr{B}})]|\!)$ are algebra homomorphisms. Their composition is an algebra homomorphism and thus it is equal to $(\!|[i', a' \circ (g \oplus FX)]|\!)$ since it is unique. □

E.3 Program Conversion

Both $h = (\!|[i, a]|\!)$ and $\mathit{hoist}\, f = (\!|[\epsilon, \iota \circ (f \otimes \Sigma^{\mathscr{A}})]|\!)$ are algebra homomorphisms. Their composition is an algebra homomorphism and thus it is equal to $ah = (\!|[i, a \circ (f \otimes X)]|\!)$ since it is unique.

$$\begin{aligned} ah &= (\!|[i, a \circ (f \otimes X)]|\!) \\ \mathit{convert} &= \mathit{free}\, \llbracket \mathit{ins}_{G\Xi} \rrbracket \\ &= (\!|[F\epsilon \circ \phi^o, F\mu \circ \phi \circ (\llbracket \mathit{ins}_{G\Xi} \rrbracket \oplus F((G\Xi)^{\mathscr{A}}))]|\!) \end{aligned}$$

We have to show that $Fh \circ F(\mathit{hoist}\, f) \circ \mathit{convert} = F(ah) \circ \mathit{convert} = (\!|[i', a' \circ (g \oplus FX)]|\!)$. We show that $F(ah) \circ \mathit{convert}$ is an algebra homomorphism $\Xi^{\mathscr{B}} \to FX$, then it is equal to $(\!|[i', a' \circ (g \oplus FX)]|\!)$ due to its uniqueness.

Proof.

$F(ah) \circ \mathit{convert} \circ \epsilon$
$=$ (def. *convert* & property $(\!|-|\!)$)
$F(ah) \circ F\epsilon \circ \phi^o$
$=$ (functor F)
$F(ah \circ \epsilon) \circ \phi^o$
$=$ (def. ah & property $(\!|-|\!)$)
$Fi \circ \phi^o$
$=$ (def. i')
i'

$F(ah) \circ \mathit{convert} \circ \iota$
$=$ (def. *convert* & property $(\!|-|\!)$)
$F(ah) \circ F\mu \circ \phi \circ (\llbracket \mathit{ins}_{G\Xi} \rrbracket \oplus F((G\Xi)^{\mathscr{A}})) \circ (\Xi \oplus \mathit{convert})$
$=$ (bifunctor $\oplus$)
$F(ah) \circ F\mu \circ \phi \circ (\llbracket \mathit{ins}_{G\Xi} \rrbracket \oplus \mathit{convert})$
$=$ (naturality $\llbracket - \rrbracket$)
$F(ah) \circ F\mu \circ \phi \circ ((F(\mathit{ins}_{G\Xi}) \circ \llbracket G\Xi \rrbracket) \oplus \mathit{convert})$
$=$ (bifunctor $\oplus$)
$F(ah) \circ F\mu \circ \phi \circ (F(\mathit{ins}_{G\Xi}) \oplus F((G\Xi)^{\mathscr{A}})) \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (ϕ is a natural transformation)
$F(ah) \circ F\mu \circ F(\mathit{ins}_{G\Xi} \otimes (G\Xi)^{\mathscr{A}}) \circ \phi \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (def. ι)
$F(ah) \circ F\iota \circ \phi \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (functor F)
$F(ah \circ \iota) \circ \phi \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (def. ah & property $(\!|-|\!)$)
$F(a \circ (f \otimes X) \circ (G\Xi \otimes ah)) \circ \phi \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (functor F)
$F\,a \circ F(f \otimes X) \circ F(G\Xi \otimes ah) \circ \phi \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (ϕ is a natural transformation)
$Fa \circ \phi \circ (Ff \oplus FX) \circ (F(G\Xi) \oplus F(ah)) \circ (\llbracket G\Xi \rrbracket \oplus \mathit{convert})$
$=$ (bifunctor $\oplus$)
$Fa \circ \phi \circ ((Ff \circ \llbracket G\Xi \rrbracket) \oplus FX) \circ (\Xi \oplus (F(ah) \circ \mathit{convert}))$
$=$ (naturality $\llbracket - \rrbracket$)
$Fa \circ \phi \circ (\llbracket f \rrbracket \oplus FX) \circ (\Xi \oplus (F(ah) \circ \mathit{convert}))$
$=$ (def. g)
$Fa \circ \phi \circ (g \oplus FX) \circ (\Xi \oplus (F(ah) \circ \mathit{convert}))$
$=$ (def. a')
$a' \circ (g \oplus FX) \circ (\Xi \oplus (F(ah) \circ \mathit{convert}))$ □